Dem Fachbereich für Mathematik und Informatik

der Technischen Universität Braunschweig

vorgelegte Dissertation zur Erlangung des Grades einer

Doktor-Ingenieurin (Dr.Ing)

von

Mojgan Kowsari

On the Development and Use of a Formal Object Oriented Methodology Based on an Application Case Study

1. Referent: Prof. Dr. Hans-Dieter Ehrich

2. Referent: Prof. Dr. Isidro Ramos Salavert

Eingereicht am 01.10.2002

Abstract

The objective of this thesis is to evaluate the object oriented specification language TROLL for industry. We used TROLL from analysis to implementation for an information system which is located at the Physical Technical Federal Board (PTB) in Germany. This information system assists different users who deal with the certification and testing of electrical equipment in an explosive atmosphere. The main part of this thesis therefore describes the advantages and the disadvantages of using TROLL in various software engineering phases and different problem domains, such as in the safety critical part. When we started this project it was clear that TROLL was not suited for all aspects which we had to deal with. However, due to the well-defined semantics of TROLL it was straightforward to extend it as needed. Limits, however, had to be accepted in certain areas such as those with real-time aspects. In this thesis, we will also demonstrate how formal techniques that include the object oriented paradigm can be made profitable in software engineering practice. Software engineers should not be asking how to use formal methods, but how to benefit from them as part of a complete software engineering approach. We will give some rules and advice based on practical experience which can provide benefits in similar settings.

Zusammenfassung

In der vorliegenden Dissertation wird die objektorientiert Spezifikationssprache TROLL im industriellen Bereich evaluiert. Hierzu wurde TROLL für die Entwicklung eines Informationssystem der PTB (Physikalisch Technischen Bundesanstalt Braunschweig) von der Analysephase bis zur Implementierung eingesetzt. Das Informationssystem unterstützt unterschiedliche Benutzer bei der Zertifizierung und dem Test von elektrischen Geräten, die in explosiven Umgebungen eingesetzt werden. Der Hauptteil dieser Dissertation beschreibt die Vor- und Nachteile von TROLL beim Einsatz in den verschiedenen Software Engineering Phasen und den unterschiedlichen Anwendungsgebieten, wie etwa im sicherheitskritischen Bereich. Beim Start des Projektes stand bereits fest, dass die Sprache TROLL nicht alle Aspekte der Entwicklung abdecken konnte. Es war jedoch aufgrund der strengen semantischen Definition der Sprache einfach, TROLL um neue Konzepte zu erweitern. An einigen Stellen, wie z.B. bei der Realzeit mußten aber die Grenzen der Erweiterbarkeit aktzeptiert werden. Zusätzlich zeigt diese Dissertation, wie eine Kombination aus formalen Techniken und objektorientierten Ansätzen effektiv im Software Engineering eingesetzt werden kann. Softwareentwickler sollten in Zukunft nicht mehr fragen, ob sie eine formale Methode benutzen sollen, sondern eher wie sie diese in der Softwarentwicklung optimal einsetzen können. Die Arbeit gibt Ratschläge und Regeln weiter, die auf den positiven Erfahrungen bei der Entwicklung des Informationssystems der PTB basieren.

Acknowledgments

I want to express my gratitude to my supervisor Professor Hans-Dieter Ehrich for having accepted me into his working group and for his constant guidance and support during the course of this work. I also wish to thank Professor Isidro Ramos Salavert for accepting the co-supervision of this thesis.

For the pleasant working atmosphere, I thank all my former colleagues of the Information Systems Group: Gabi Becker-Würch, Jutta Bleiß, Grit Denker, Christiane Eberhardt-Herr, Silke Eckstein, Antonio Grau, Peter Hartel, Juliana Küster Filipe, Thomas Mack, Karl Neumann and Ralf Pinger.

I am especially grateful to Juliana Küster Filipe and Antonio Grau who accompanied and supported me greatly during all stages of this thesis.

Grit Denker and Peter Hartel for their support and good team work during our project time.

The analysis and implententation of the project at PTB was carried out by more than 20 students doing their student and diploma theses.I want to thank them for their excellent work. In this context, I especially want to acknowledge Mohamed Battikh.

I also want to thank Maren Krone, Uwe Arnold, Hamid Shafiee for their motivation and support during the writing of this thesis.

In addition, I would like to express my gratitude to my present colleagues at Lufthasnsa Revenue Services, Department for Information Managing, for their support, patience and friendship.

For reviewing the English in this thesis, I want to express my appreciation to Petra Besemann.

I also wish to thank my parents, my husband and my best friends for their love, encouragement, understanding and support. I dedicate this thesis to them.

Finally, I am indebted to the Physical Technical Board for their financial support of this work.In this context, I would like to express my thanks to Dr.Uwe Klausmeyer for having made this thesis possible. Without his enthusiasm and his visions, this project would never have been realised.

Contents

List of Figures

Chapter 1

Introduction

Using formal methods helped us to build the right system and helped us to build it right - at no extra cost.

- Anthony Hall, IEEE, 1996 [Hal96]

This chapter will discuss the motivation, the context, the objectives and the structure of this thesis.

1.1 Motivation

Information systems are reactive systems capable of maintaining and utilising large amounts of data. Most safety critical systems, such as banking systems and railway systems, belong to this category. Information system development is an iterative process which begins with the definition of a system requirement, continuing with design refinement and ending with its implementation. Early error detection in system requirement specifications reduces the effort and the time spent with corrections in other software process phases[Wes02, BAM+02].

In 1996, more than US $180 billion were spent in the United States on software development projects which had failed because their requirements were insufficient, elusive or changed and the domains were not properly understood [FME97]. For this reason, application of a formal method in an industrial environment is taking an increasingly central and important position in the development of complex information systems [HB95, ABL96, BP01].

At present, it is difficult to specify what a software system will do. On the one hand, most software systems include different subsystems: understanding the interaction of these subsystems as well as understanding the system within its environment are only a few of many important aspects that are necessary to describe system specification. On the other hand, requirement capture is not easy: the software system has various different users each of whom is only familiar with his own subjective evaluation of the system. Consequently, no one has a full understanding of the entire system [JBR99]. Thus,developing and specifying complex software systems correctly is an important area of research in the computer science community. Required is a formal language at a high level of abstraction which does not permit ambiguities. There are two approaches associated with the above activities: the formal method and the object oriented method.

The formal method is based on a mathematical notation which precisely defines requirements and eliminates residual errors in the specification. The object oriented method concentrates on methodological aspects which when combined with a graphical notation improves the communication between users and developers as well as the system set-up of the abstract model.

A specification language can be regarded as a way of writing a contract between customers and system designers. A certain degree of specification clearness and readability is necessary for domain specialists. Formality is also necessary in order to attain additional information for further design and implementation.

Moreover, modelling of complex information systems requires the use of an approach which covers both the static and the dynamic aspects of the system on a high level of abstraction.

In the object oriented paradigm a system is viewed as a community of interacting objects which allows for a detailed presentation of real world entities and can reflect their behavioural and static properties. However, popular object oriented methods such as UML [RJB99] lack the formality required to model complex information systems. From a practical point of view, it is advantageous to combine both approaches in order to develop complex information systems.

The formal method has established itself in the area of safety critical systems, while the object oriented method is used for application in business. The fact that a complex information system can include different problem domains makes such a combination attractive [EH96]. The users of the formal method try to enhance their approach with the object oriented method while the users of the object oriented method try to utilise the formal method to analyse their semantics [Lan95, GK96]. Several formal specification languages already have object oriented extensions, e.g. VDM++ [DK92], MooZ [MC90]. Although they have been adapted to the object oriented method, they must still contend with a low level of abstraction.

Case studies are needed to demonstrate the benefits of such combinations. Such studies are easier to plan than other techniques because they are purely empirical and/or psychological. However, they are more difficult to interpret and cannot be generalised [KPLP95]. A case study can show the effects of technology in a typical situation, but it cannot be generalised and applied to other possible situations. Therefore, systematic evaluation is needed to understand the strengths and weaknesses of each method, their appropriate contexts and tasks to which each is most suitably applied.

The central questions are: which parts should be evaluated and how? By answering these questions, we can learn more about the use of existing methods and how to engineer new methods.

1.2 Context

The work found in this thesis was developed in the context of the formal object oriented specification language TROLL 3.0 (Textual Representations of an Object Logical Language) [DH97, Har97, GKK+98]. TROLL is a language designed for the analysis and design of distributed information systems. The roots of TROLL can be found in articles mainly devoted to semantic foundations of object oriented specifications [SSE87, SFSE88, EGS90, ES90, SJE92, EDS93, EJDS94, SHJE94].

These articles were the starting point for the design of a series of specification languages based on the object paradigm. The language OBLOG was presented in [SSE87, CSS89, Esp93]. The language TROLL [JSHS91, JSHS96, HSJ+94, HKSH94] was developed in the following years based on OBLOG.

The design of the third and current version of TROLL has been significantly influenced by the experience gained in an industrial project located at the PTB (Physikalisch Technischen Bundesanstalt: Physical Technical Federal Board) in Braunschweig [Kow96, HDK+97, SK97, KG98]. The work reported in this thesis was started in 1994 and has been mainly supported by the PTB. Theoretical foundations, distributed logic [ECSD98, EC00], module theory [Küs00a, Küs00b] and model checking [EP00, PE01] are addressed. Work towards extending TROLL by a module concept is carried out by [Eck98, Eck01].

The study and development of tools supporting the modelling and animation of TROLL specifications is done by [Gra01]. Besides further application projects in co-operation with the PTB, TROLL are being applied in a project which aims at combining the TROLL and Petri net approaches to software specification in a railway traffic control application [EG01].

1.3 Objectives

The objective of this thesis is to apply a formal object oriented language called TROLL in a case study. As already mentioned above, the project is a cooperation with the Physical Technical Federal Board (PTB) located in Braunschweig. This project began in 1994 and was completed in 2000. We used TROLL for analysing, specifying, and implementing an information system to help different users to deal with the certification and testing of electrical equipment in an explosive atmosphere Figure 1.1). The experience that we gained from this project improved our understanding of how to specify TROLL. In this thesis, we present a method for specifying with TROLL and illustrate the use of the language concepts by means of examples. We used TROLL in different software engineering phases and problem domains, for example the safety critical part. With the help of an example from the safety critical part we will show step by step how TROLL and its graphical part OMTROLL are used in system analysis, design and implentation in our case study.

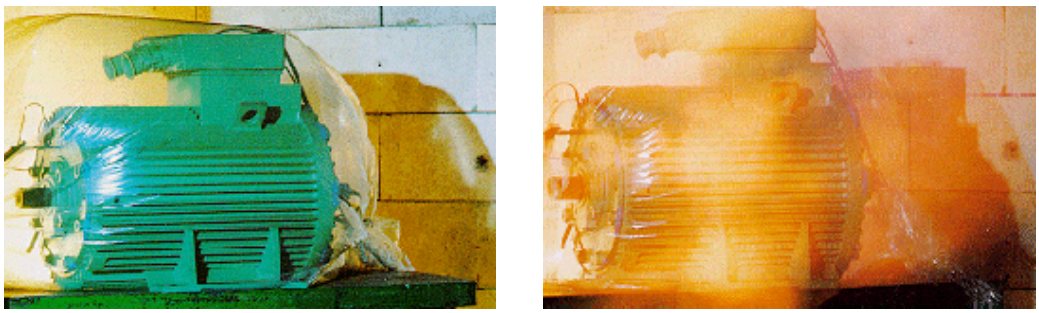


Figure 1.1: Flame Proof Enclosure

1.4 Structure of the Thesis

The thesis is structured as follows: **Chapter 2** will compare different approaches in the area of methodology. Emphasis will be placed on the use of formal object oriented methods in the real world. Most formal methods only consist of a specification language which is based on mathematical notations and sometimes a test and validation system. In the last 25 years, a lot of research has been carried out in this area. However, the methodological aspect for the development process has been repeatedly neglected. It is therefore very important to introduce methods which deal with this aspect.

After a short introduction showing the different software lifecycle models, this chapter will describe two different categories of method. The first is based on one specification language with its own environmental systems and development processes and will be introduced by summarising the concepts of Albert II, OBLOG, OASIS and UML. The second category is created as a compilation of different specification languages and their environmental structures. This will be illustrated by the Heisel Approach and BOOAD.

Chapter 3 illustrates our case study, introduces the problem domain and the information system to be developed. The system description is presented graphically using OMTroll˙

Chapter 4 presents modelling with Troll and its graphical part OMTroll. The language concepts are introduced by using examples from CATC. In order to illustrate the use of Troll in the design of the information system and development of our methodology, we will focus on one part of the CATC system, namely the remote controlling of valves (VENTIL). We will describe the modelling of VENTIL step by step using our methodological guidelines. This chapter concludes with a brief description of the tools contained in the Troll workbench and describes how and when it was used in CATC.

Chapter 5 will highlight our results and important aspects of using Troll in our case study. We will first present our development process and procedure as well as the team structure. After briefly introducing the phases of the software lifecycle in our project, the chapter will describe the benefits of using Troll in the implementation phases. We will focus on a translation from Troll to C++ and show how the Troll specifications can be mapped into C++ - as well we will show a set of mapping rules. Finally, the chapter will concentrate on experiences concerning the integration of various system parts of CATC.

Chapter 6 will sum up the main contributions of this thesis and suggest some directions for further work.

Appendix A shows the syntax of OMTroll and Troll. **Appendix B** contains the Troll specification of the example used throughout this thesis. **Appendix C** shows the structure of Object winodws Library . Finally, **Appendix D** contains the C++ classes of the example used for metrics in Chapter 5.

Chapter 2

The State of the Art and Related Work

2.1 Introduction

> *The question software engineers should now be asking about formal methods is not whether to use them, but how best to benefit from them as part of a complete software engineering approach* *Anthony Hall, [Hal96]*

In this chapter, we will compare different approaches for methodological aspects in the development process. Emphasis will be placed on the use of formal object oriented methods in real world applications. In the development of complex real world applications various process steps are needed. Apart from notations and graphical interfaces, procedures and rules on how to apply these languages adequately become essential.

Experts do not want to deal with the entire notation because they feel it is too complex. It can therefore be very helpful to offer experts a requirement analysis manual which includes further refinements - from the planning phase to the implementation phase. Most formal methods are only a specification language possessing one mathematical notation and in some cases a test and validation system.

In the last 25 years, a lot of research has been carried out in this area. The methodological approach, however, has repeatedly been neglected in the development process. It is therefore very important to introduce methods that can solve this problem. There are basically two different categories of method. The first is based on one specification language with its own environmental systems and proceedings and will be introduced in section 2.3 by summarising the concepts of Albert II, OBLOG, OASIS and UML. The second category is a compilation of different specification languages and their environmental structures. This will be described by the Heisel Approach and BOOAD in section 2.4.

According to the above categories, a distinction is made between the formal approach which allows an object oriented extension like VDM and approaches that are formally object oriented such as AlbertII. Before we begin to describe different methods in sections 2.3 and 2.4, the question of lifecycle models must be discussed. Lifecycle models are an important part of software development and various lifecycle models exist, e.g. the waterfall model, the spiral model, the prototype model and the evolutionary model.

In section 2.2 the definition of methods will be examined. It is very important to find an unanimous definition on which we can look back upon in chapter 4.

2.2 Software Lifecycle Models

It is important to set-up a software development plan whenever a software development project is started. Such a plan is made up of different parts: resource and schedule estimates, organisation and staffing, validation and verification etc. [BH96]. One of the important elements in such a plan is a lifecycle model for organising and managing the software development process. A software lifecycle model is an abstract representation of how software is developed and includes concurrent or sequential phases in the software development process [BH96, MS92]. Lifecycle models are therefore crucial parts of development methods, however, they have not yet evolved sufficiently to support new paradigms such as object orientation [Hes97]. Nevertheless, there have been many activities in this area in the last few years and we will try to give an overview of these in this section.

Generally, a software development model refers to the whole system in regard to requirement analysis, system design, software implementation and testing. In the last 30 years, a number of lifecycle models have been developed which describe the ideal development process. These models should be regarded as a guideline for project development, e.g. the traditional waterfall model, the fountain model, the spiral model and the prototype model. In practice, the waterfall model has become the standard model. This model shows the different phases in which software projects are developed: each phase starts with a number of queries which have arisen in the previous phases. The aim is to achieve a fixed number of results. During these phases, large systems are divided hierarchically into smaller components. These components are then developed, tested and integrated separately into the system. The integrated system is then installed into the project surroundings and after acceptance released for use. This waterfall model, however, has not been accepted because the development of complex systems does not flow linearly. For this reason, gradually integrated models have been developed, such as the incremental model, the spiral model (one of the most popular models) and finally the evolutionary model.

The spiral model (Figure 2.1) was developed by the experiences made with refinements on the waterfall model [TD96]. The spiral model consists of cycles which involve a progression through the same sequence of steps: from an overall concept of operation down to the coding of each individual programme for each portion of the product and for each of its levels of elaboration.

The development of object oriented concepts made people think more intensively about how a software lifecycle should be created. Development processes are too complex to be shown linearly. Some parts of the analysis may be carried out before the design begins while other analysis parts may proceed parallel with design and implementation of other parts of the system.

In 1988, Brian Foote introduced the concept of the Fractal Model [MS92]. The fractal model tries to characterise the phases of these iterative activities and describe how mature reusable components result from them. The fractal model refines iterative notions developed in the waterfall model, the evolutionary development model and in particular the spiral model (Figure 2.2).

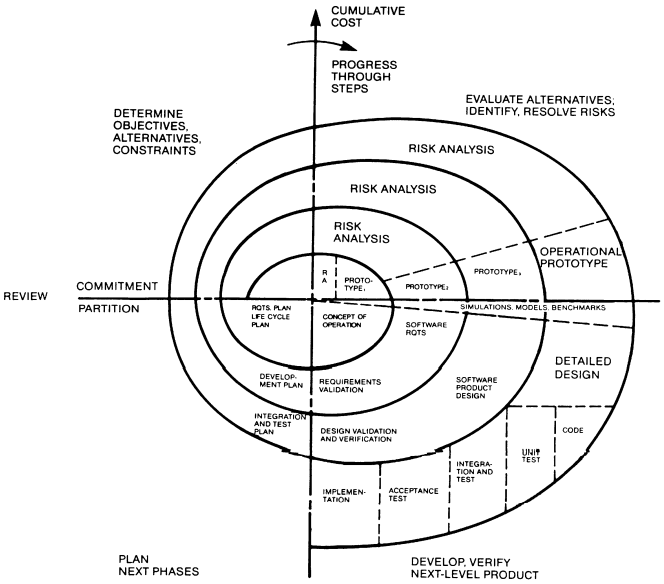


Figure 2.1: Spiral Model

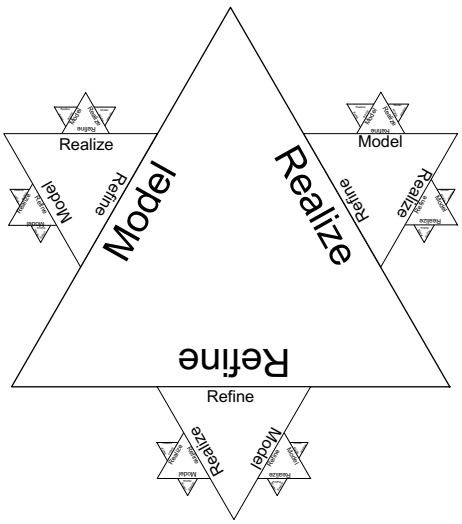


Figure 2.2: Fractal Model

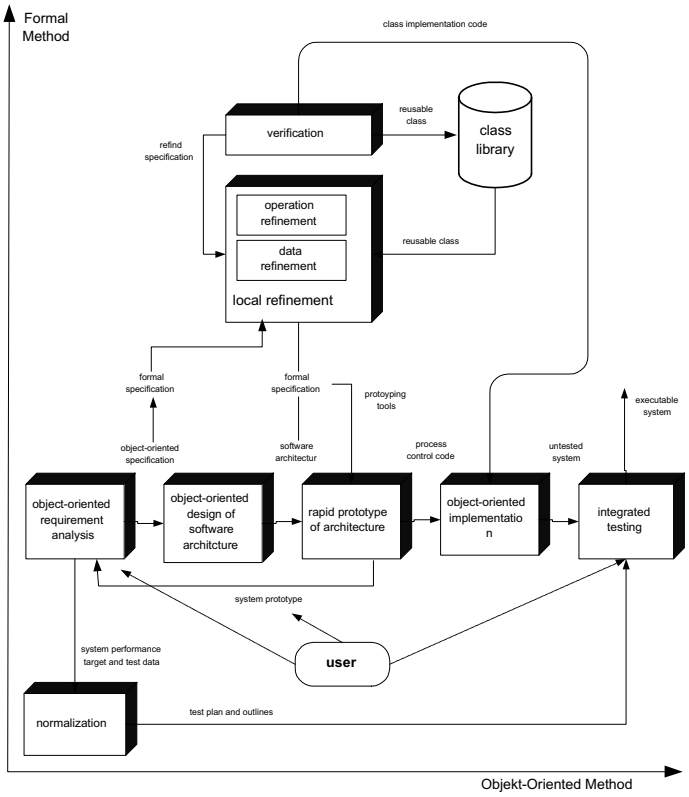


Figure 2.3: Two-Dimensional Model

McGregor and Sykes [MS92] combined these concepts with the philosophy of object oriented software development. The three different corners of the triangle stand for

- MODELLING the problem domain
- REALIZING classes
- REFINING and REUSING the resulting product.

Each corner of this triangle may repeatedly be refined further.

All of the depicted models are linear one-dimensional models. However, different approaches can be used, namely the formal method and the object oriented method to realise information systems with the help of two-dimensional models [XJG98]. In the two-dimensional models, the object oriented method is used as the horizontal thread and the formal method is used as the vertical thread. The horizontal thread describes the software process through analysis, design, prototyping, validation, implementation and testing. The vertical thread shows the software process progress in formal description, formal specification, data refinement, operation refinement, verification and further refinement (Figure 2.3).

The lifecycle model which was discussed last is referred to as an Evolutionary Object Oriented Software (EOS) development [Hes97]. It ties development cycles and activities into software development components and deals with management and project planning from a practical point of view (Figure 2.4).

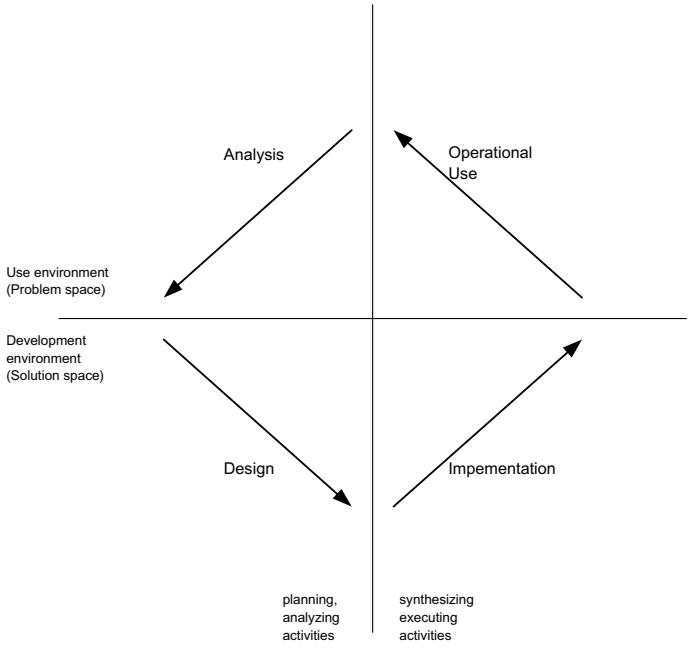


Figure 2.4: EOS Model

The fractal model appeared to be the best development process for our project because it has the same structure that our project wants to realise. This is why we made our method conform to this model as described in chapter 4. It should be emphasised that a software process description hardly stands a chance of success without determining a lifecycle model beforehand.

2.3 Definition of Method

It is an important activity in the software engineering community to build high quality software systems which are not over budget in a short period of time. Problems increase exponentially with the size of the system. The main problem in the software development process is understanding the problem domain at an early stage, namely the analysis phase.

One of the many difficulties lies in understanding the customer's needs and wants. Applying a formal specification language to system requirements early in the development process can result in more leverage. Errors discovered late in the development process are much more expensive to rectify than those discovered in the early development process. The important characteristics of a software requirements specification to solve this problem are described in [BH96]. A requirement specification must

- separate whats from hows
- be precise
- be unambiguous
- be understandable

- be consistent
- be modifiable.

Nevertheless, a specification language alone is not enough to develop a complex information system. We need guidelines and rules based on a strong notation that show us how we are going to develop our system step by step in different development phases. In the last few years, much effort and energy has been invested to develop such a methodology. A software development methodology can include an integrated set of software engineering methods, policies, procedures, rules, standards, techniques, languages and other methodologies for all development phases [Bjo98].

Companies must invest to train their staff to learn different methods. More complex methods need more investigation and more investigation means higher costs and more time. Especially small and medium sized companies are interested in short-term solutions due to their limited budgets. What can be done in this situation? How can a method be made less complex? These are two questions which we will try to answer. In this context, it is important to mention that "less complex" does not necessarily mean "fewer concepts".

To begin with, it is important to give a clear definition of method. A commonly used understanding of method describes it as consisting of a notation (textual and graphical) and a set of guidelines and examples showing how the notation can be deployed [Ste94, Rum94]. This description does not provide any hints as to how to manage complexity. Rumbaugh provides a more comprehensive definition of method which consists of:

- A set of fundamental modelling concepts
- A set of views and notations for presenting the underlying modelling information to humans
- A step by step iterative process for constructing models and implementation of them. *The process may be described at various levels of detail, from the overall project management down to the specific steps to build low level models*
- A collection of hints and rules of thumb for performing development. These are not organised into steps and may be applied wherever they make sense.

This definition supports multiple views of different aspects of a system, allowing for hints which may only apply within certain domains.

Requirements must be made of how method definitions by Rumbaugh may be applied and explained in more detail. A methodology is more likely to be used when it is simple, clear, effective and small. The most confusing aspect of most methodologies is understanding the large number of notations that must be applied. The questions asked are normally as follows:

- Which concepts are available in our notations?
- How are the different concepts related?

- When can a notation be used?
  - In which phases of the lifecycle?
  - Which parts of the system can be described with it?

Due to these problems, it is important to define method characteristics to make them suitable for the real world as follows [TD96, FL98]:

- Guide the development of a system all the way from customer requirements to testing
- Include both notations and process descriptions
- Specify phase products such as documents and figures
- Allow extensions and be easy to learn and use.

2.4 Methodologies Based on a Unique Approach

In this section, we will give an overview about methodologies which are based on an unique approach. We will distinguish between the formal object oriented approach, the formal approach and the object oriented approach.

2.4.1 Albert II

The formal object oriented specification language AlbertII [DB95] was developed at the University of Namur (Belgium) for specification of functional requirements for a concurrent real time system. An AlbertII specification is structured in terms of agents which can be described or defined as basic units. Specifications are made up of three sections:

- The definition of abstract data types and operations on them
- The definition of societies, i.e. how agents and societies are grouped together to form a hierarchy
- The definition of the agents.

AlbertII concepts are named agents, but do not differ in their significance from TROLL objects and their attributes, respectively. Components are called state components while the definition of actions is similar. AlbertII makes a distinction between actions of a certain duration and instant actions without duration (instantaneous actions). AlbertII defines certain constraints to confine possible behaviour. It is based on an object oriented variant of temporal logic (Albert-CORE). Constraints are used for pruning the infinite set of possible lives of an agent which result from the graphical declaration of its structure. The life of an agent consists of a sequence of alterations and conditions; the condition being the period between two successive alterations. Each alteration has an attached automatic time stamp.

Alterations may be called up by three types of events:

- the start of an action
- the end of an action
- the happening of an instant action.

Constraints are part of an agent declaration and will be formulated with the help of predefined templates. These templates are syntactic simplifications for certain types of expressions of temporal logic. While an attribute or an action in TROLL 3.0 may basically be declared invisible for other classes by detailing hidden information, co-operation constraints of AlbertII may express more flexible combinations as the relevant visibility may be made dependent on a temporal condition, i.e. it will be redecided in every condition. The semantics of AlbertII are given by mapping AlbertII specifications into fragments of the Albert-CORE logic which has been strongly inspired by Real Time Object Specification Logic. A tool set is designed for assisting the analyst in his task of analysing a requirements specification written with the AlbertII [Hey97, HD98]:

- AORE (Agent Oriented Requirements Specification Environment)
- KBRA (Knowledge Based Requirements Assistant).

In practice, experiments have shown that it was not possible to support calculations (possibly automated) for deducing properties from a system description for AlbertII specifications. In order to solve this problem, automated and intuitive support is provided – a conceptual reasoning assistant for exploring specifications and gaining an initial understanding of them before it is possible to reason formally about them. If the effort should pay off in the experimental process, errors may already be disclosed. A distinctive characteristic is to use automated techniques in a local manner whenever it is appropriate. This corresponds to the early stage support for analysing AlbertII specifications.

AlbertII defines several strategies which are followed by the analyst. These strategies should not, however, be interpreted as formal rules. They are applied in practice in a flexible way and are often combined together.

- A "Goal Oriented" Strategy
- A "Retracting Assumptions" Strategy
- An "Agent Oriented Framework" Strategy.

The first two strategies are based on progressive refinement of the specification while the last strategy emphasises the possibility of reusing generic specification components.

AlbertII is applied in many **Case Studies**, namely in computer integrated manufacturing, in telecommunications, in control systems for a tower crane, video on demand and in satellite communication systems [DDBZ95].

2.4.2 OASIS

In comparison with other approaches, OASIS (Open and Active Specification of Information System) Semantics for OASIS specifications are given by a set of logical formulas expressed in an extension of dynamic logic which are interpreted by Kripke structures. In the last few years, emphasis has been placed on the animation of OASIS specifications in concurrent environments [LSR99]. An animation environment based on concurrent logic programming is presented in [LSR97, Let99]. In this environment, specifications are first modelled in a graphical editor and then stored in a repository. Specifications are then translated into KL1 concurrent logical programs which are compiled to obtain a prototype that can be carried out. A graphical user interface allows users to simulate events that can also be read from a file and observe the actions occurring in the objects and the reached states. In the OASIS animator, objects communicate with one another asynchronously while in the TROLL animator, they communicate synchronously, i. e. communication entails a synchronisation between the lifecycles of the participating objects. Unlike the TROLL animator [Gra01], the OASIS animator does not support inheritance, aggregation or integrity constraints, although it is supposed to support them as well as synchronous communication in up-coming versions. OASIS developed a CASE environment supporting object oriented methods based on OASIS [PIP+97, PPIG98, PCR99]. The object oriented method CASE allows developers to specify graphically the system, using OASIS as an intermediate language to automatically generates a code in several programming languages. Unlike the TROLL animator, the code generation in the object oriented method CASE is not oriented towards validation purposes, but rather to obtain the final application.

2.4.3 UML

The Unified Modelling Language (UML) is an object oriented graphical language for analysis and design. It essentially unifies the Booch, Rumbaugh (OMT) and Jacobson (OOSE) methods, emerging from a cooperation between their designers, and has become a standard modelling language [FK97, RJB99]. It is unified across:

- historical methods and notation
- the development lifecycle
- application domains
- implementation languages and platforms
- development process
- internal concepts.

The relationships between various UML diagrams are given in (Figure 2.5) [HW98]

The Unified Modelling Language (UML) was adopted as a standard by the Object Management Group (OMG) in November 1997 and has undergone gradual development since then. UML covers all phases of software development; beginning with the analysis through to implementation.

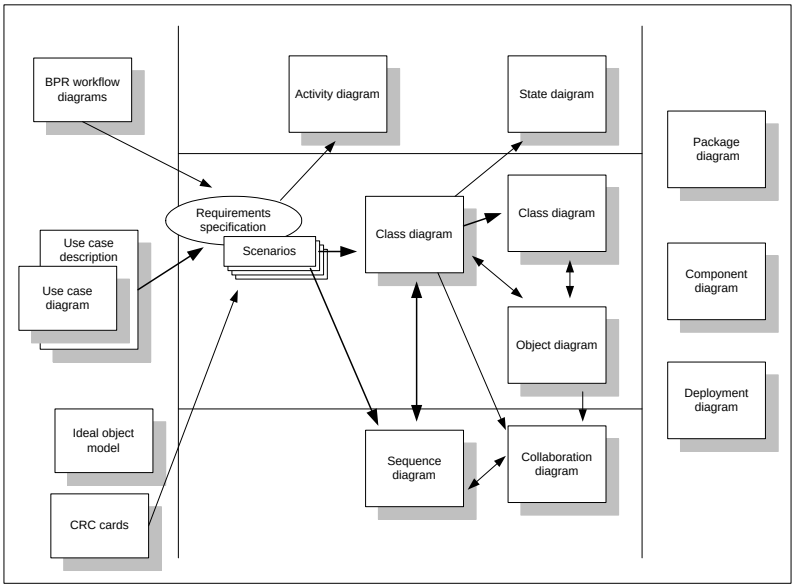


Figure 2.5: UML Overview

UML consist of numerous different groups of diagrams which may explain various aspects of the system to be developed from different perspectives at different times within the development process:

Use Case Diagrams will be placed at the beginning of a requirement analysis and present a rough and informal view of the limits of the system.

Static Structure Diagrams (Class Diagrams) are found in all phases of the analyses and of the development in different detailed stages. The class diagram explains the static structure of the system with the help of object classes, its signatures as well as the different relationships between these classes.

Object Diagrams show snapshots of a system at a certain point in time, i.e. the concrete instances of the classes from the class diagram that exist at the given point of time.

Sequence Diagrams are one of the two interaction diagrams offered in UML to describe the interactions between objects. Sequence diagrams accentuate the temporal sequence of messages.

Collaboration Diagrams contain essentially the same information as sequence diagrams but shown in a different way. They emphasise the structure and the relationships between the objects involved in the interaction.

State Diagrams (state chart diagrams) explain the accepted transition of object conditions and their behaviour in form of condition automations which consist of conditions, transitions, events and activities.

Activity Diagrams are based on condition automations, however, they only possess activity condition and triggerless transitions. They are used to explain workflows and operations.

Package diagrams are a further structural diagram which permits the logical division of the classes to be shown in modules.

Component Diagrams are understood as the implementation of a limited system part with fixed interfaces; usually containing multiple co-operating object classes. Component diagrams may show the co-operation and the dependencies of

Figure 2.6: Activity

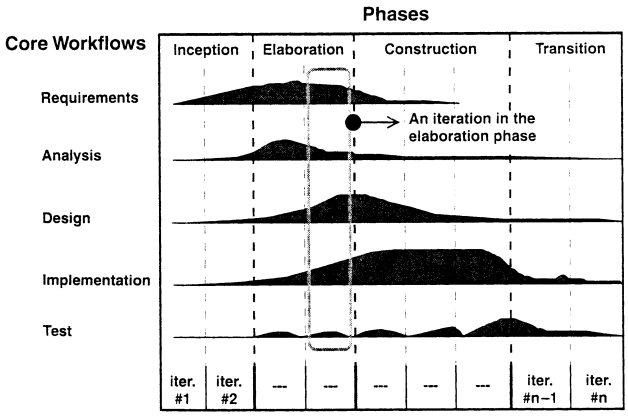


Figure 2.7: Development Process

several components.

Deployment Diagrams are the second type of implementation diagrams. With their help, the configuration of a realised system may be shown in order to divide the components into different nodes.

The Unified Development Process [JBR99] uses UML as a language to describe different activities in development processes. It defines a software process as a set of activities needed to transform user requirements into a software system (see Figure 2.6). The unified process is component based: the software system being built is made up of software components connected via interfaces. Moreover, it is characterised as

- use-case-driven: the users' requirements are captured in use cases
- architecture-centric: the system architecture is developed to meet the requirements of key use cases
- iterative: in that the project is broken down into mini projects (one for each iteration). In each iteration some part of the system is analysed, designed, implemented and tested
- incremental: each of the previous parts are an increment and the system is built incrementally.

Figure 2.7 shows how the workflows requirements, analysis, design, implementation and test take place over the four phases, namely, inception, elaboration, construction and transition. The curves approximate the extent to which the workflows are carried out in each phase. Recall that each phase usually is subdivided into several iterations which constitute mini-projects [JBR99].

2.4.4 OBLOG

Object Logic (OBLOG) is a further formal specification language [SSE87]. OBLOG is the result of research work started in connection with the EC ESPRIT project IS-CORE and has been developed into a commercial product by the OBLOG software company in Lisbon. Based on the above mentioned publication [SSE87] OBL-89 was developed and explained in [CSS89]. The definition of an abstract object type was introduced based on abstract data types in this publication and for the time that the OBLOG was used as a language to specify a society of integrated objects. OBL-89 distinguishes itself from other languages by structure object systems and different language constructions to specify certain object characteristics. To specify the object dynamics, a process description language leaning on CSP was introduced with the option of including safety conditions to specify live conditions for an object.

OBL-89 specifies object instances which are in accordance with genetic object descriptions (templates) through regulations of an identification mechanism (surrogate). To structure objects, concepts like specialisation, generalisation and aggregation are introduced. Parallel to the definition of OBL-89, a first tool support has been worked on and the prototype of a specification animator has been developed. An in-part circumstantial syntax and several not very intuitive language constructions have lead to no practical realisation of OBL-89. The two direct follow-ups of OBL-89 were OBLOGwb and TROLL1.

Tools used for OBLOG are Editor, Animator and Designer. They support the tasks of analysis and design, automate mechanical tasks such as codes and documentation production and aid in validation and verification.

2.5 Methodologies Based on Different Approaches

2.5.1 Heisel Approach

The Heisel Approach [Hei97] gives methodological support for the application of the formal method. The emphasis lies in making the formal technique applicable for non-experts. This approach defines agents and strategies as steps that support some software development task which need collaboration between experts on formal techniques and those parties who will be applying formal techniques.

An **Agent** is a list of activities which may depend on each other when carrying out some tasks in the context of software engineering. The description of activities are informal (see Figure 2.8).

Strategies serve to formalise a wide variety of software engineering activities and is a generically known representation for software development activities. Strategies are understood as concepts which are formally defined

Figure 2.8 shows different steps defined in this approach:

- Define all relevant notions of the application domain
- Define the requirements for the system to be built
- Convert the requirements in a pragmatic way into a formal specification
- Set-up mapping between the requirements and the formal specification

No.	Phase	Validation
1	Define all relevant notions of the application domain.	
2	Define the requirements for the system to be built.	Every important aspect of the application domain and the system must be expressible. For each of the defined notions a statement for the system schould be made.
3	Convert the requirements in a pragmatic way into a formal specification.	All relevant aspects of the system must be expressed appropriately. the specification must be more abstract than code.
4	Set up a mapping between the requirements and the formal specification.	Each requirement must show up in the specification. Each part of the specification must belong to a requirement.
5	Validate the specification.	Besides inspecting the specification use a many of the following mechanismus as appropriate: checklists, animation, proof of properties, testing.

Figure 2.8: Agenda for Specification Acquisition

- Validate the specification.

The Heisel approach can be used for every formal language.

2.5.2 Method for the First Project

This method is developed by Nokia Telecommunication, Finnland by A.Jaaksi [Jaa98]. The notation of the simplified method includes three main elements:

- Natural language
- Class diagrams
- Sequence diagrams.

Figure 2.9 illustrates the five phases of this method. Although the phases are listed sequentially, a repetitive phase can be adopted as well. There are two parallel paths in the process of system development: the static path uses class diagrams and the functional path uses operation descriptions.

2.5.3 BOOAD

Business Object-Oriented Analysis and Design (BOOAD) [Bay99] describes an object oriented analysis and design methodology that is tailored to software systems whose functionality is mainly implemented by database operations. The notation and semantics are inherited from OMT. Additionally, BOOAD contains notation derived from database languages as well as programming languages. For example, *queries*, *transaction*, *integrity*, *exception mechanism*. The process of BOOAD can be described as follows:

- identify the outward behaviour of the system by discovering actors and how each actor interacts with the system
- specify the user interface

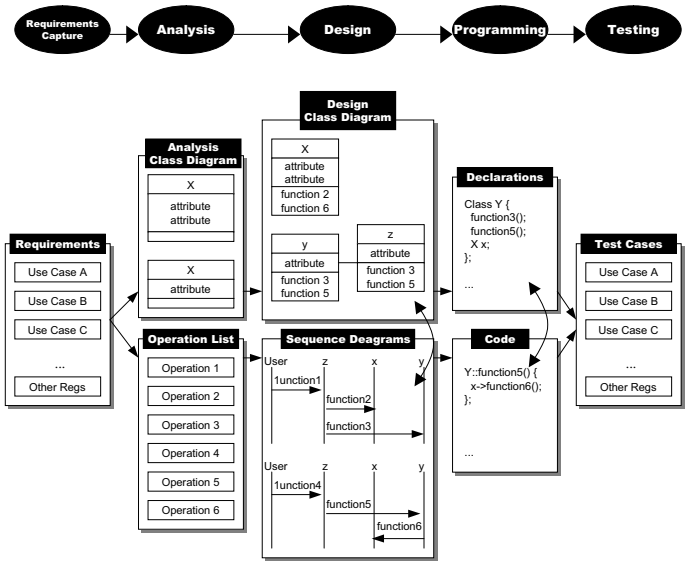


Figure 2.9: First Method Overview

- specify the static structure of the system
- specify the dynamic behaviour of the system
- map user interface events to system events
- verify the model.

Chapter 3

Case Study: The CATC System

3.1 Problem Domain

Our project was carried out at the Physikalisch-Technische Bundesanstalt[1](PTB) in Braunschweig/Germany. [JB87]. The PTB builds the foundation for scientific, technical and legal meteorology and of physical safety engineering and exercises control functions in these fields. Its tasks are laid down in laws and regulations. Approximately 2000 employees work in 10 divisions in the fields of research, measurement and consulting.

Group 3.4 "Explosion Protected Electrical Equipment" works with the testing and certification of explosion proof electrical equipment. Its research is based on the European Standard EN 50014-50028 [EN 87a, EN 87b]. Equipment which has been approved and certified by the European Standard is permitted to be set up in hazardous areas. The tests guarantee the *safety* standard required by specifying construction features and operating conditions for devices which may constitute a hazard.

The assessment procedure is carried out by 80 employees responsible for testing and certifying different electrical equipment in 3 laboratories. All necessary steps are carried out manually by the staff in charge and are individually worked out. In 1994, Group 3.4 started a co-operation with the database group of the Technical University of Braunschweig in order to computerise the certification procedure and thus improve the communication among the PTB employees as well as between PTB, manufacturers and 15 other similar laboratories in Europe.

It is very important that certification procedure information can be reused and made available at any time. CATC (Computer Aided Testing and Certifying) is the software system which was employed.

Due to the importance of safety factors in the certification procedure, the reliability and robustness of the system takes a central position in the development of CATC. The certification procedure steps are not necessarily sequential. Sequences of the individual actions can partially be carried out on a parallel basis by an officer.

To automate these steps, an information system was developed in which the entire system was partitioned and separately processed into the following part systems:

- the administration system ADMIN

[1]Federal Institute for Science and Technology

- the standard system ExPert
- the standard system for plastics ExPlast
- the system PRESSTEST
- the system VENTIL
- the system JONIT.

The very complex certification procedure consists of preliminary screening of all formal and informal documents and verification of the design papers. These are, for example, technical drawings and experimental tests such as explosion tests, flame propagation tests and thermal-electrical investigation tests which are all carried out according to the European Standard.

3.2 System Description

The process that electrical equipment must go through during the certification procedure can be described in the following steps:

- registration of application of the manufacturer
- assessment of the design papers for the equipment based on descriptions and its accordance with the European Standard
- preparation of the check lists, including all the necessary experiments that must be carried out
- setting up the experimental tests and documenting the results
- preparation of the settlement of the account
- issuing a certificate or refusing an application depending on the test results.

There are two types of users involved in the above procedure: *staff* and *operators*. The first three steps and the two last steps are carried out by the staff, while the operator is responsible for experimental testing. During the certification procedure, the communication between staff and operators is extremely important. Both have access to the test results. Operators produce and store the test results while the staff interprets them.

Test results which belong to a certain experiment can be pre-selected and accessed by the staff. The operators start the experiment. There are different tests depending on the kind of electrical equipment used, i.e. the pressure test for a motor can be described as follows: a prototype (2) is placed inside a test chamber (1) which is called an *auto clave* of an *explosion test stand* and is filled with an explosive atmosphere (eA). A spark (S) then ignites the atmosphere inside the enclosure (see Figure 3.1).

A prototype passes the test if the enclosure withstands the developing pressure and temperature (°C) and the explosion does not continue into the autoclave. 30

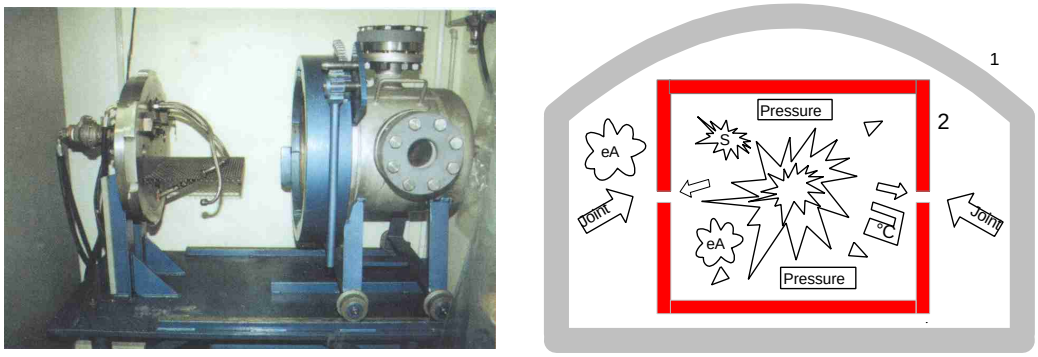


Figure 3.1: Pressure Test.

kbytes data are produced every 0.2 seconds during the explosion, thus resulting in a conflict between hardware which works in real time and the operating system.

In order to avoid any errors, the European Standard requires that every application is tested by more than one experiment. While the experiments for one application are being carried out, the staff may set up the next experiment.

CATC has a safety-critical application which is composed of a technical aspect and a database aspect in a heterogeneous complex environment. At the same time, it takes already existing and re-developed applications into consideration.

The above mentioned steps can be categorised as follows:

1. support *experimental test*,

2. manage *basic administration data* and

3. allow for *design approval*.

CATC has access to the central database of the PTB via LAN where common data is stored. Further programmes for administration (RBEZ[2], HASY[3]) access this database (see Figures 3.2, Ex-Link). CATC is not a stand-alone information system: it must be embedded into an existing environment. Furthermore, existing application programmes which have to be re-specified because they were erroneous must be dealt with: these re-specified sections must be embedded into the new information system structure. There is also a link to the multiply accessed PTB-wide database. Figure 3.2 illustrates the hierarchical structure of the intended information system.

3.2.1 Basic Administration

The *administration* management which includes such information is essential for the following tests in the certification procedure and must permanently be available.

[2]archive and documentation application
[3]settlement and calculation application

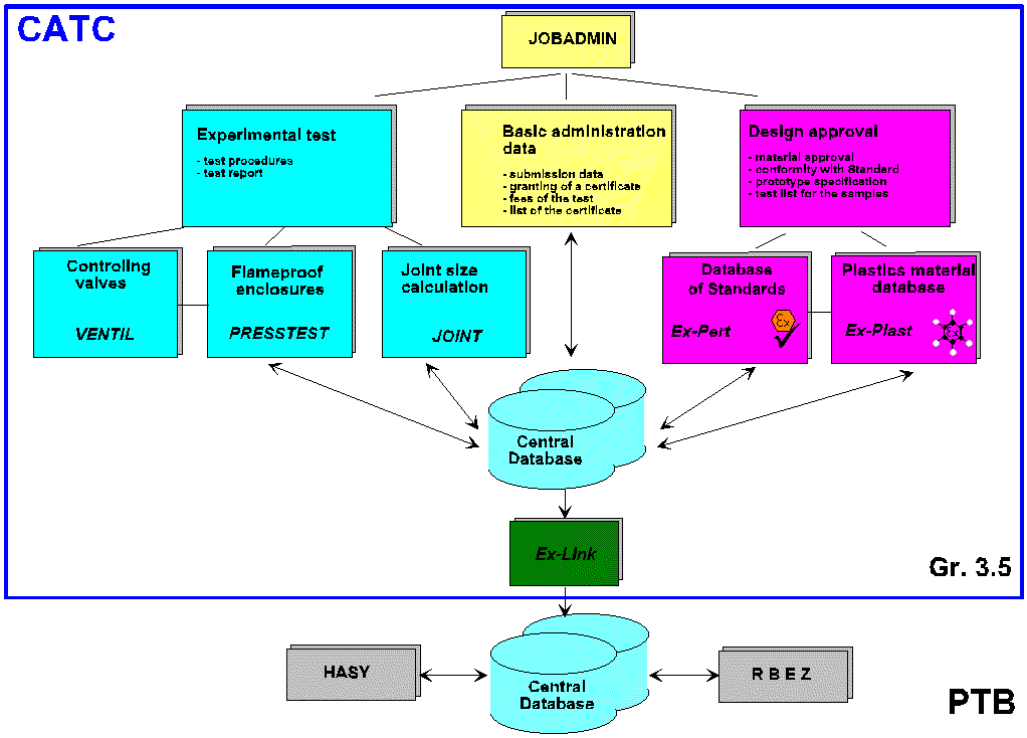


Figure 3.2: CATC Overview

3.2.2 Design Approval

The subsystem dealing with *design approval* provides the relevant clauses of the European Standard so that the certification procedure can become more efficient. Thereby, required data can be retrieved more quickly and easily at every desk.

3.2.3 Experimental Tests

The subsystem for the *experimental tests* stores all relevant data which are performed by the operators in the test laboratory. The aim is to ensure that flammable parts which potentially can ignite an explosive atmosphere are placed in an enclosure. This enclosure can withstand the pressure which develops during an internal explosion of an explosive mixture and can also prevent the spreading of the explosion to the explosive atmosphere surrounding the enclosure.

3.2.4 CATC Modelling

Specification of the CATC system is made up of four *nodes*: the user node, the hardware node, the information system node and the timer node. (see Figures 3.3)

The information system node (in the following abbreviated as IG34) and the system user are modelled into two separate nodes. This is essential so that more than one user can work simultaneously with the IG34 System as a user model, i.e. as a component within the IG34 System which would result in only one user being able to work at a time.

The IG34 node presents the passive part as well as the data interface of the system. The desired user behaviour could be specified in the active part of the node

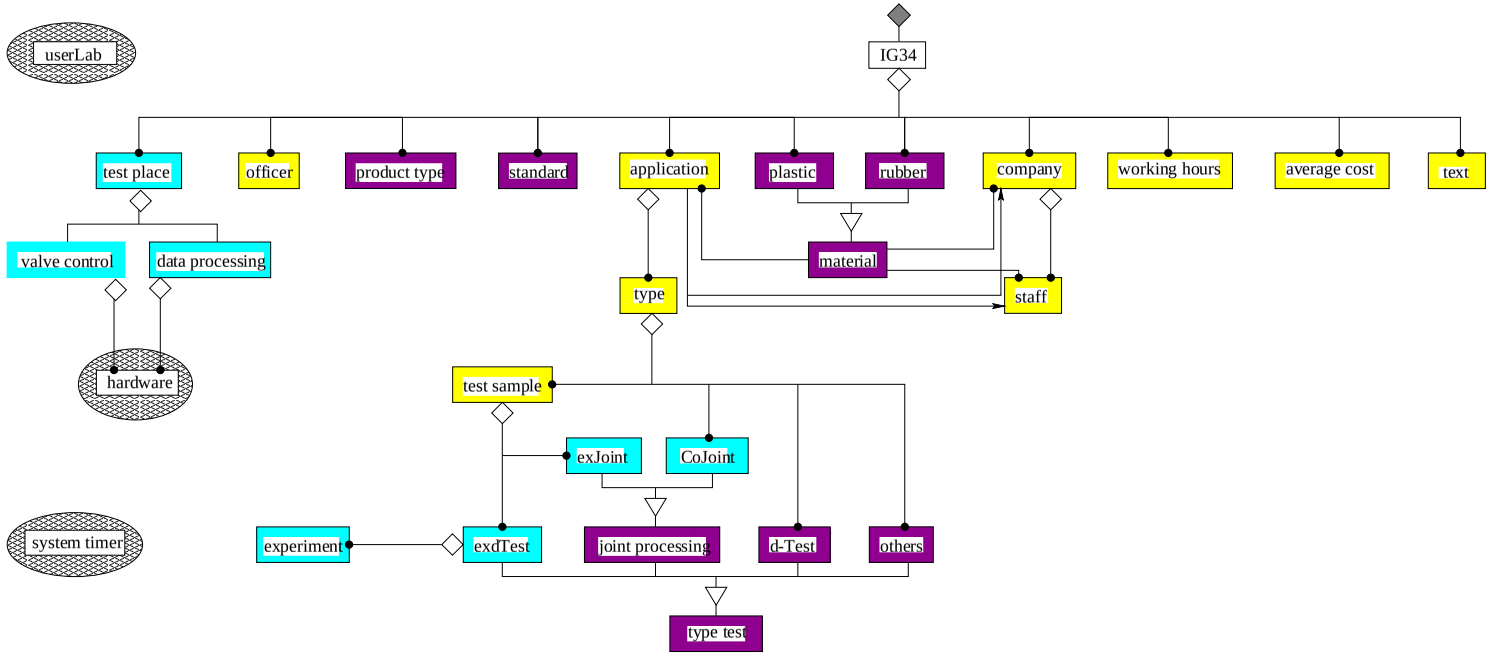


Figure 3.3: IG34 Overview

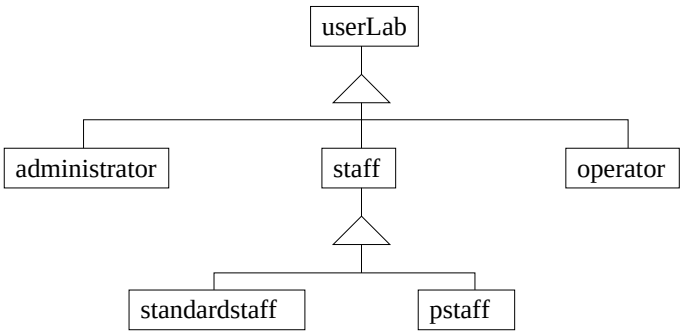


Figure 3.4: USER Overview

behaviour and build the user interface.

The user node describes the possible behaviour of the different user groups such as testers, technicians etc. and their interfaces to the main system. Digital outputs such as "open valve", digital and analogue sensors, voltage representing measured pressure etc. are modelled in the hardware node.

The IG34 and user nodes communicate with one another by means of an interaction, i.e. a user who wants to enter or delete data in the database by means of the user interface will call up events in IG34 by interaction.

When the hardware close parts of the systems were modelled, it was clear that the real existing hardware had to be made simultaneous to the two other nodes. For this reason, a further node test stand hardware was created. The course within the node test stand hardware and between IG34 and test stand hardware partly underlay real-time requirements and the specification language.

Since TROLL does not dispose of a language construction, a further node system timer was created. With its help, the real-time requirement was modelled.

The IG34 system and how the separation work was specified showing all part systems is illustrated (see Figure 3.2). Each part depicted in can be described by different objects (see Figure 3.3). For example "Basic Administration Data" requires *object application, object type, object company etc.* We showed the object in Figure 3.3 with the same colours as in the Figure 3.2.

The object class `Group` is called a *node* of the system which is concurrent to the other node shown in Figure 3.3, namely system timer as well as user node. Furthermore, the IG34 system users were divided into different user groups whose relationship to one another is shown in (see Figure 3.4).

Object class `User` is specialised in `Staff`, `Operator` and `Administrator`. Object class `Staff` is specialised in `pStaff` who deal with certifying of electrical equipment with plastic parts like switches and `standardstaff` who deal with other electrical equipment like motors.

The system users can only call their components via the node IG34 as the user interface and the database interface were modelled in different nodes. When object class IG34 disposes events to manipulate the instances of its components, they will be called by events of the users.

we will now take a look at the specification of a whole object system (see Figure 3.5). The keyword `object system` followed by the name of the system is the opening bracket. The specification itself consists of four parts: data type and object class specifications, object declarations and a behaviour part where the global interactions are described. We gave the name CATC to the system. We do not consider any user defined data types.

```
object system CATC
  data type ...
  object class Application ... end;
  object class ...
  objects IG34:Group;
  objects Users(userId:nat):User;
  behavior
  Staff(Users(userId)).createExperiment(appNr, nam, st, expNr)
     do
        IG34.Applications(appNr).newExperiment(nam, st, expNr)
     od;
end.
```

Figure 3.5: Object System CATC

Two kinds of objects are described: `IG34` is the name of an instance of object class `Group`. After the occurrence of the accompanying birth actions, this object will also contain objects of class `Application` as well as of class `Experiment` as components. The declaration of several objects of class `User` is done by parameterisation, i.e. each instance is identified by a `userId` of type `nat`. As `User` is the base class of a specialisation hierarchy, the objects will either have a special aspect `Staff` or `Operator`. In the behaviour part, we give an example for a calling from an object of class `Staff` (identified by the `userId` of the base class) to a certain object of class `Application` which can be reached via the aggregating object `IG34` through "dot notation". Whenever the action `createExperiment` occurs in `Staff`, the action `newExperiment` in `Application` is called which in turn calls the birth action of class `Experiment` (see Figure 3.6).

The community diagram contains the aggregated object class `Group` which represents Group 3.4 of the PTB. It has object classes `Application, Company` and `Standard` as components, whereby `Application` itself has `Experiment` as such. Aggregation is represented by a diamond and the dot at the top of some of the object classes denotes multiple components. For example, each application may own a number of accompanying experiments, locally identified by their experiment number (`expNr`). We wil now take a look at object class
application. The object class `Application` as shown in Figure 3.6 has several `Experiments` as components, each one is identified by a unique experiment number. It contains four attributes: `company` is an object valued attribute, i.e. a link to object class `Company`. In contrast to components, objects referenced by attributes are not embedded in the classes where the attributes are declared.

The second attribute `labour` has a user defined data type `labours` which is not specified here. `appl_date` is a constant attribute which means that once the data is set, it can never be changed again.

The third attribute `nextExpNr` is initialised with one and is hidden, i.e. it is only visible inside the object class `Application`. It stores the next number used to

identify the experiments belonging to this application. Furthermore, there are two actions: `createAppl` is a birth action that creates an object of class `Application`. It has all attributes of the class as parameters, excluding the fourth attribute`nextExpNr` which is automatically initialised. `newExperiment` calls the birth action of the component class and increases `nextExpNr`. Finally, a constraint assures that there are not more than nine experiments for each application.

```
object class Application
  components
   Experiments(expNr:nat):Experiment;
  attributes
   company : |Company| constant;
   labor : labors;
   appl_date : date constant;
   nextExpNr : nat initialized 1, hidden;
  actions
   *createAppl(comp:|Company|, dat: date, lab:labors);
   newExperiment(nam: string, st:msset, !expNr: nat);
  behavior
   createAppl(comp, dat, lab)
      do
        company:=comp,
        appl_date:=dat,
        labor:=lab
      od;
   newExperiment(nam, st, expNr)
      do
        Experiments(expNr).createExp(nam, st),
        expNr:=nextExpNr,
        nextExpNr:=nextExpNr+1
      od;
  constraints
   nextExpNr<10;
end;
```

Figure 3.6: Object Class `Application`

Now we will take a look at another part of the system, namely joint process. First we briefly explain some *technical notions* which are necessary for the understanding the specification. *joint* (see Figure 3.1) is the place where corresponding surfaces of two parts of an enclosure come together and prevent the transmission of an internal explosion to the explosive atmosphere surrounding the enclosure [HO71].

In Figure 3.1 we will illustrate a test surrounding for joint tests. The main components to measure and estimate joints according to the standard given are the *width* and the *gap of a joint*. The width of a joint is the shortest distance from the inside to the outside of an enclosure. The gap of a joint is the distance between the corresponding surfaces when the electrical apparatus has been assembled. The prototype tests on flame proof is comprised of tests on the ability of the enclosure to withstand pressure and of tests on the non-transmission of an internal ignition. Therefore, the enclosure is placed in a test chamber called autoclave and some explosive mixture is introduced into the enclosure.

The European Standard specifies the design of flame proof joints in detail. During the testing procedure it is important to compare the standards values of the widths and gaps of the joints with the applicants value resulting from the explosion

Joint	Data acc. to EN 50 018 - 1977/ VDE 0171			all to construction drawing						test sample		
	Minimum length of joint	Distance l (for bores within flamepath)	Maximum width of joint i_K (flat, combined, cyclindric, bearinggap and actuationsgap)	Width of joint L (c+d)	Distance l (a+b)	Size of diameter with ISO-symbol for internal and outside measurement	Dimensions (acc. to DIN 7160, DIN 7161)	Constructional width of gap = difference of diameter between bore and cyclindric bore: min.	Constructional width of gap = difference of diameter between bore and cyclindric bore: max. = i_c	Length of joint L	Distance l	Width of testing gap i_E
K1	12.5		0.15	140	–	$\frac{7.5+00.5}{7.45-0.05}$		0.05	0.15	15.42	$\frac{7.57}{7.333}$	0.225
K2	12.5		0.15	141	–	$\frac{7.5+00.5}{7.45-0.05}$		0.051	0.148	15.43	$\frac{7.57}{7.333}$	0.227

Figure 3.7: Table of Flame Path Joints

tests.

Process of `jointTest`

There are two groups: staff and operators who can manipulate joints. The applicant, i.e. the person who wants some device to be certified sends the table of flame path joints to thr PTB (see Figure 3.7). There are three different kinds of values:

- Columns 2-4 give the data according to EN 50018
- Columns 5-8 include data according to construction drawings
- The last three columns of the table are values resulting from tests.

The staff compares the data according to EN50018 with construction drawing and decides whether the values are satisfactory. The operators verify the values provided by the applicant and report their results to the staff. The staff is responsible for the assessment of the values measured by the operator.

A part of the community diagram of the CATC system (see Figure 3.3) is depicted in Figure 3.8. It consists of the object classes `JointNode, JointTable, Joint, ExpJointPart, ConstJointPart` and `JointPart` (the same Object class
tt joint processing as in Figure 3.3).

`JointNode` is the object class that models the special part of CATC concerning joint tests. In this universe, we have joints and joint tables. We simplified the specification to one joint table and several joints. The diamond stands for aggregation of objects and the triangle is the diagrammatic notion for specialisation. Thus a `JointNode` object is an aggregation of one `JointTable` and one or more `Joints`. A `JointTable` has a list of joints as attributes. This is modelled as an

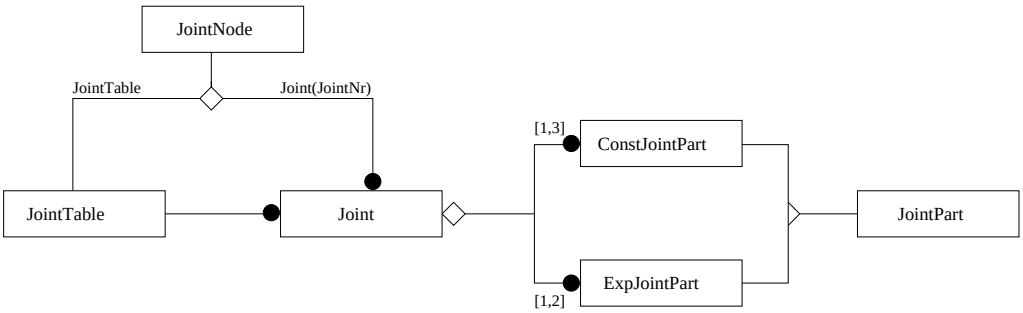


Figure 3.8: Fragment of the Object Community Diagram of CATC

object-valued attribute. Joints can be constructed of several parts, constructive parts (`ConstJointPart`) and experimental parts (`ExpJointPart`). As already mentioned, the constructive part is concerned with comparing data according to the standard with data according to the construction drawing. The experimental part deals with the results of test measurement. Up to five parts can belong to one `joint` which form one row in the table of flame path joints (see Figure 3.7). There may be one to three constructive parts and one to two experimental parts. The object class `JointPart` depicts a specialization, which consists of those attributes and actions, the constructive and experimental parts have in common.

The interfaces of the objects specified in the comunity diagram are specified in Figure 3.9.

Joint
cons: map (range (1,3)) to ConstJointPart esp: map (range (1,2)) to ExpJointPart
row : /derived
* create

Figure 3.9: Object Declaration Diagram of `JointPart`

For each object class an attribute and an action alphabet has to be specified. The attributes represent the observable state of an object of that class. Actions model the operational interface of the objects. The actions are declared by a classwide unique identifier and a list of parameters. Actions that create new objects are called birth actions and are indicated by an *. The life of an object is terminated with the occurrence of a death action. Such actions are marked with an +. The longterm behavior of objects is specified in Figure 4.3).

Staff objects are born by login and by it they are in the “NoJointTable” state. This is the beginning of the lifecycle of a staff object. It may logout immediately after login and by this will leave the system. Logout is the *death* action of a staff object. After a login a staff object may build a joint table. By this action the life cycle state changes to the “JointTable exists” state. Now it can work with the joint table, e.g. print it or do other things not specified here. From this life cycle state, a staff object may logout or remove the joint table. The latter action will change

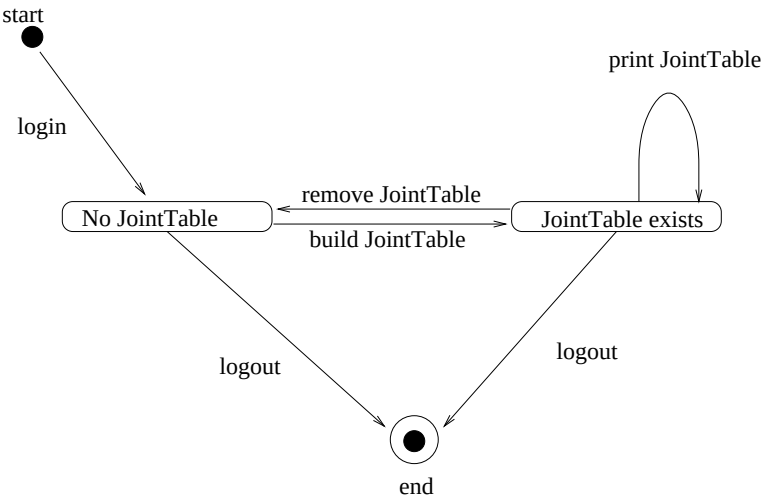


Figure 3.10: Object Behavior Diagram of the Staff

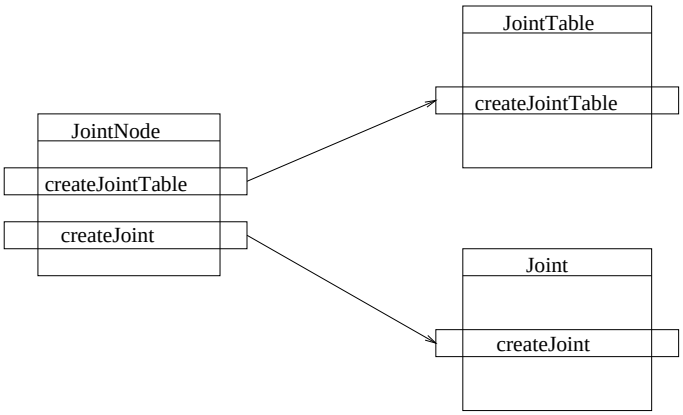


Figure 3.11: Object Communication Diagram between `JointNode`, `JointTable` and `Joint`

back to the life cycle state where no joint table exists.

The communication between `JointNode` objects and `JointTable` and `Joint` objects is depicted in Figure 3.11. The action `createJoint` of a `JointNode` object causes directly a create action of a `Joint` object. The action `createJointTable` of a `JointNode` causes a create action of a `JointTable`.

Now will take a look at textual specification of our object classes. We started with the object class `JointNode`.

```
object class JointNode
  attributes
    JNr: nat initialized 1;
  components
    Joint(nr:nat) : Joint;
    JointTable : JointTable;
  actions
     * new;
     createJoint;
     createJointTable;
```

```
  behavior
    createJoint
      do Joint(JNr).create; JNr:=JNr+1 od;
    createJointTable
      do JointTable.create od;
end;
```

There are two components: A joint node has several joints which are identified by a number (`Joint(nr:nat)`) and a joint table (`JointTable`) as components. Moreover, an attribute `JNr` is specified to save the current joint number. The initial value of the attribute is `1`. Thus, after a `JointNode` has been created by `new`, `JNr` has value `1`. There are two actions specified, one for creating joints another one for creating a joint table. The former one takes the current joint number `JNr`, increments `JNr`, calls synchronously the birth action in `Joint`, and assigns the new joint to the name `Joint(JNr)`. These birth actions are specified in the corresponding object classes, i.e. `JointTable` and `Joint`, respectively.

```
object class JointTable
  attributes
     Joints : list(Joint);
  actions
     * create;
     insertJoint(j:Joint);
  behavior
     insertJoint(j:Joint)
       do Joints:= append(Joints,j) od;
end
```

The object class `JointTable` has a list of joints as attributes. These are object-valued attributes. In contrast to components, object-valued attributes do not belong to `JointTable`, instead they are readable from `JointTable`. In this sense, object valued attributes are links to other objects such that their attributes can be read and used for some computations or their actions can be called. Besides the creation action, there is one action specified to append new joints to this list, i.e. to append further links. `insertJoint` is the action which takes a joint as parameter and appends this joint to the list `Joints`.

Before we specify `Joint` we introduce the object classes `ConstJointPart` and `ExpJointPart`, as well as the generalization of both `JointPart`. The object class `JointPart` comprises all attributes which are also part of the specialisation. Every joint has a `gap`, a `width`, and further attributes named `a`, `b`, etc. See the table of flame path joints in Figure 3.7 where these attributes appear. The creation of a `JointPart` is parameterised by the necessary values for the joint attributes and an indication, if the joint shall be an explosion joint or a constructive joint.

```
object class JointPart
   attributes
      gap : real;
      width : real;
```

```
      a : real;
      b : real;
        ...
    actions
      * create(kind:enum(constructive,explosion),
                    gap:real,
                    width:real, a:real, b:real, ...)
      ...
end;
```

The object classes `ConstJointPart` and `ExpJointPart` are specialisations of `JointPart` which have further attributes. In principle inheritance in TROLL is monotone. This means, that we carry over all axioms of the superclass into the subclass. The behavior of the subclass is in full compliance with that of the superclass. For our case study, we specialize `JointPart` to `ConstJointPart` which has an additional derived attribute:

```
object class ConstJointPart
  aspect of JointPart if create(kind, gap, width, ...)
```
kind = constructive; *attributes*l : real *derived*(a+b); *actions... end*

Now we come to the object class `Joint`:

```
object class Joint
  components
    cons(1..3) : ConstJointPart;
    exp(1..2) : ExpJointPart;
  attributes
    row : list(record(a: real, ...,  gap:real, l: real))
              derived concat(toList(select jp.a, ..., jp.gap, jp.l
                                    from jp in range(cons)),
                             toList(select jp.a, jp.b, ..., jp.gap, 0.0
                                    from jp in range(exp)));
  actions
    * create;
      ...
end;
```

An object of class `Joint` has up to five components. Three components are constructive joint parts and another two are experimental joint parts. The former ones are those which will be derived from the construction drawings, whereas the latter are fixed by explosion test done by the operators in the laboratories. There is an attribute called `row` for joints. This attribute corresponds to one row of the table of flame path joints in Figure 3.7. The sort of this attribute is quite complex because in `row` the information of all components are collected. We specified a `select` statement to extract this information and in this way exploited the logic calculus which provides concepts for querying object states. We explained this by starting from the innermost select clauses: the select clause returns a bag of records. Each record incorporates five real numbers representing `width`, `gap`, etc. of one

joint. We query the constructive joints as well as the experimental joints. We get all joints by the implicitly defined operation `range`. `Range` gives the set of all elements of the co-domain of the declared component, i.e. all existing component objects. Here, `range(cons)` delivers all constructive joint parts. We selected the values of the attributes and transform the bag to a list. To concatenate experimental and constructive joint part list, we introduced 0.0 as `l` (length) value of `ExpJointPart`. The result of this concatenation is one `row`.

Until now, we only specified object classes. Thus, we still have no object instances. These will be generated by declaring the system's objects. We simplify our sets of instances such that we have one object of class `staff` and one object of class `JointNode`. The object identifiers may be parameterized. Thus, we can declare infinite sets of objects. The TROLL specification of `staff` corresponds to the behaviour diagram in Figure 4.3.

```
object class staff
  actions
    * login;
    newJoint;
    buildJointTable;
     ...
end;

objects JN:JointNode end;

objects user:staff
   behavior
     newJoint
       do JN.createJoint od;

     buildJointTable
       do JN.createJointTable od;
end;
```

We showed a simplified part of specification of CATC, the whole specification of THE CATC system is carried out in [Goe97, Him97]. Especially the hardware specification is given in [Hoh96, HS94b, Sha96]. The adminstration part is carried out in [Saa96].

Chapter 4

Methodological Concept of TROLL

TROLL is a formal object oriented specification language for modelling and designing of distributed information systems. It incorporates many ideas based on experience made in developing the **OBLOG** family of languages and their semantic foundation [SSE87]. TROLL1 [JSHS91] is a dialect which has its roots mainly in an early textual version of **OBLOG**. It was formerly known as oblog$^+$. TROLL*light* [GV96] is a simplified version of TROLL with an operational process language for object life cycles. TROLL*light* is available with a development environment offering special tools for verification and validation. TROLL2.0 [HSJ$^+$94] has several restrictions with respect to TROLL1 such as the absence of roles [DHS94]. TROLL 3.0 which will later be discussed here is devoted to modelling distribution issues. It has been designed with the aim of being executable. The TROLL approach supports the declarative specification of conceptual models. It integrates concepts of dynamic modelling aspects, structural aspects and process aspects and is based on a *Distributed Temporal Logic* (DTL) [Ehr96, DE97, ECSD98] which is a special temporal logic that describes properties of distributed objects. In the last eight years, much work has been carried out on these theoretical foundations [SSE87, ESS88, EJDS94, DE95, ES95] and its methodological issues [SJ92, SJH93a, HJ95, HKSH94, JSHS96]. TROLL allows for a descriptive modelling of complex systems [HJS92, HJSE92, HJS93b, SH94] and incorporates concepts from the object oriented method, conceptual data modelling and process modelling [JHS93, HJS93a, HS93, HHKS94, SHJ$^+$94]. Aside from the textual notation, a graphical syntax was proposed in [WJH$^+$93, JWH$^+$94]. The use of this diagrammatic presentation allows an easy transition from informal to formal specification documents. This has been the prerequisite for a wide acceptance of TROLL as a software engineering method [SJH93b, HS94a, HJ94, HJ95, ZH94]. A set of tools to support the TROLL method has been developed and integrated into a workbench [Gra01]. In this chapter, we first introduce TROLL 3.0 and its graphical notation. Next, we present the guidelines supporting the modelling with TROLL and explain them by example. Finally, we briefly show the workbench.

4.1 Introduction to TROLL Version 3.0

Structure of Specification Documents.

A specification document describes a system world and is named an **object**

system. An object system consists of the following basic elements:

- Definition of problem specific data types
- Specification of object classes
- Declaration of system objects
- Specification of system global interaction rules.

Any number of appearances of these elements and in any sequence order may be found within a system specification.

4.1.1 Data Types

Data types definitions are used in TROLL to build new data types from already existing ones. Among the data types predefined in TROLL we can distinguish between simple and complex data types.

Simple Data Types

Simple standard predefined data types in TROLL are *bool*, *nat*, *int*, *real*, *date*, *char*, *string* and for which the usual operations are available. With the help of data type constructors, new problem specific data types can be defined. The most simple constructor is the enumeration type, $enum(id_{12}...id_n)$ whose values are shown by the symbols $id_{12}...id_n$.

Example:
data type **type of material***enum*(plastic, elastomer)

The only operation for enumeration types is the conformity test. Further problem specific data types may be defined by the specification of an identifier coupled to an existing data type. The identifier may be used within the specification as an alias to the type. Another constructor available in TROLL is the record type, which allows the definition of data tuples.

Example:
data type **address***record*(street : string,
house no. : string,
postal code : nat,
place : string,
country : string)

Complex Data Types

TROLL allows the use of constructors to build complex multi-valued data types. These constructors are *list(t), set(t), bag(t), map(id:*t_1*,*t_2*)*. For a more detailed description of these constructors and their manipulation operators we refer the reader to [DH97].

Data terms are built by variables, attributes and constants which may be combined through operators defined for their corresponding data types. A very useful

data term is the query term: "*select* term *from* range *where* condition" which is similar to the query term defined in SQL.

Example:

select e.name *from* e *in* **rng**(employee) *where* e.address.place="Braunschweig"

This query returns a set with the names of all employees living in Braunschweig.

4.1.2 Object Classes

The basic elements of a system to be described are objects. An object class represents the abstract description of similar objects. An object class is described by an object class specification. It contains an interface or signature declaration and a behaviour definition. Part of the signature is the declaration of attributes and actions: attributes indicate the characteristics of the objects while actions define the operational interfaces of an object. The behaviour corresponding to the actions is defined in the behaviour part. An object class declaration has the following form:

object class **Name**
```
        [specialisation condition]
        [components component declaration]
        [attributes attribute declarations]
        [actions action declarations]
        [behaviour behaviour declarations]
        [constraints constraint conditions]
```
end;

Object classes may be complex. They may be specialisations of other classes (*is-a relationship*) or have other classes as components (*part-of relationship*). The behaviour of an object is defined by the description of rules for the actions and the declaration of integrity constraints. Next we will describe each element in further detail.

Attributes

All attributes are listed after the keyword **attributes**. An attribute declaration contains the name of the attribute, its data type as well as optional characteristics. The following attribute characteristics are permitted in TROLL:

- initialized
- constant
- derived
- hidden
- optional.

These characteristics may only be used for attributes and are not available for local variables.

Actions

Actions change the state of an object by altering the value of the attributes. They represent the basic element for specifying the allowed object development. All action declarations are listed after the keyword **actions**. An action declaration contains the name of the action and an optional parameter list. Output parameters are denoted by the symbol "!" in front of the action name, birth actions by the symbol "*" while death actions are given by the symbol "+". All other actions are considered update actions. By using action calling, the objects have the ability to communicate with each other and to observe certain rules. All actions which are marked with the key **hidden** can only be accessed locally.

Aggregation

Objects containing other objects are called complex objects. Each component object can only belong to one complex object. Components do not have an independent existence. They depend existentially on the general object to which they belong. For single components, only one instance of a component can exist for a complex object. For multiple components, the different component instances are uniquely identified through parametrised names. Complex objects may have access to all attributes and actions of their components.

Example:
object class **IG34**
components applications (appl.no.):application
...

end;

The object class *IG34* has the object class *application* as a multiple component. With the component name application and the parameter*appl.no.* for the component, various instances of application can be utilised.

Specialisation

Objects may also be declared as *specialisations* of other objects. A specialised class represents a particular aspect of a basis class. For the specification of a specialisation class, a basic class and a full list of specialisation conditions must be given. The specialised objects inherit the attributes and events of the general objects and can own further attributes, actions and constraint conditions. The subclass declaration is provided with an additional keyword **aspect** which refers to the superclass.

Example:
object class **material**
...
actions *initialize plastic (Wdata: initmaterial
*initialize elastomer: (Wdata: ...)

...
end;

object class **elastomer**
aspect of material if initializeelastomer ...
end;

object class **plastic**
aspect of material if initializeplastic ...
end;

In this example, the object class *material* is the superclass of the subclasses *elastomer* and *plastic*. *Material* defines two different birth actions: one for each specialisation class. The subclasses specify the respective birth actions as specialisation conditions.

Object Life (Behaviour)

The life of an object starts with a birth action and ends with a death action. Before the birth or after the death of an object, its characteristics can not be observed nor can its actions be called because the object is in a non-defined condition. An object class may declare several birth or death actions, but during an object life, only one birth and death action may happen.

The following example shows the object class definition of the *explosion test stand*. It has objects of classes *measurement system* and *remote controlling of valve* as single components. For every instance of the *explosion test stand*, one instance of *measurement system* and one instance of*remote controlling of valve* exist.

Example:
object class **explosion test stand**
components
measure: measurement system;
controlling: remote controlling of valve;
actions
∗ set-up (conf.: explosion test stand configuration);
+ remove;
behavior
set-up (conf)
do measure:SET_UP(conf.measure),
controlling.set-up(conf.controlling)
od
remove
do measure.remove
controlling.remove
od
end;

Constraint Conditions

The **constraints** part of a class declaration establishes integrity conditions which restrict the possible values of attributes. TROLL makes a distinction between initial and static constraints: initial conditions are checked only at the birth of an object, static conditions must be fulfilled in each object state.

4.1.3 Declaration of System Objects

An (*object system*) consists of a quantity of objects which belong to object classes. All interactions between objects of different object hierarchies are specified outside the classes, in the system specification part which is initiated by the keyword **behaviour**. In the area of communication, the access to attributes and actions of different TROLL object classes is strongly limited. An access is only allowed in a hierarchic direction, i.e. from top to bottom. TROLL version 3.0 makes a distinction between two types of communication:

1. communication between complex objects and its components
2. communication between concurrent objects (system nodes).

The communication between a complex object and its components is specified within an object class. Aggregated objects have access to actions and attributes of its components, but not in the reversed order. Communication between concurrent objects refers to objects which exist on a parallel basis and are independent of each other.

4.2 The Graphic Notation OMTROLL

This section explains the graphical presentation of TROLL specifications. The advantages of a graphical presentation lie mainly in a better illustration of complex and global relationships. A graphical notation is especially adequated for giving a first overview of the system and for communication between users and developers. In order to give graphical support to TROLL a notation called OMTROLL was developed. OMTROLL combines concepts of TROLL with graphic elements of the popular *Object Modelling Technique* (OMT). The language elements of OMTROLL are divided into five different diagrams:

1. Community Diagram
2. Object Class Declaration Diagram
3. Object Behaviour Diagram
4. Communication Diagram
5. Data Type Diagram.

In the following, we will describe briefly each of these diagrams.

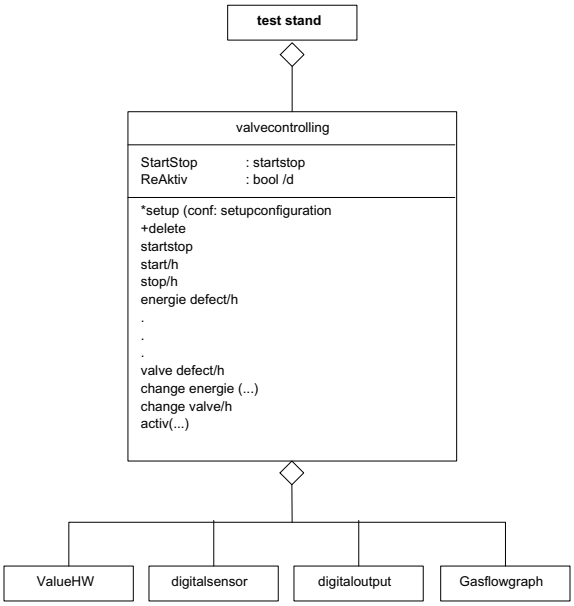


Figure 4.1: Class Declaration Diagram

4.2.1 The Community Diagram

The community diagram gives an insight into the object classes of a system and their relationships. It illustrates the static structure of the system. An object class is represented by its name put into a box (see Figure 4.1). Component and specialisation relationships are denoted by diamonds and triangles respectively.

4.2.2 The Object Class Declaration Diagram

An object class declaration diagram must be designed for every object class of the community diagram. It contains all attributes and actions which refer to the object class. The diagram consists of a box divided into three areas: the class name, the declaration of attributes and the declaration of actions. Object declaration diagrams and the community diagram are usually represented together. Figure 4.1 gives an example of the IG34 system.

4.2.3 The Communication Diagram

The communication diagram shows the communication structure between the objects of the system. Figure 4.2 shows an example for the communication between the object class IG34 and their text block and product group components, e.g. the action store product group of the complex object IG34 calls up the action create of the product group.

4.2.4 Object Behaviour Diagram

One can explicitly define the life cycle of an object by object behaviour diagrams. Figure 4.3 shows a behaviour diagram. Nodes in the diagram represent the possible states of an object. Edges between nodes represent state transitions and are labelled

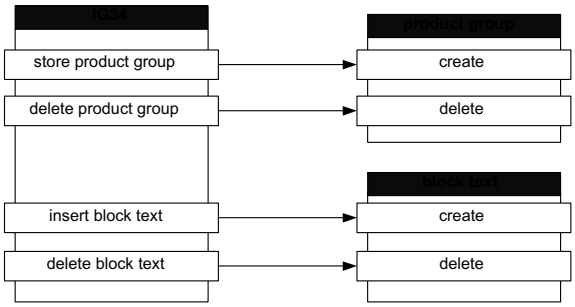


Figure 4.2: Communication Diagram

with the action that causes the state transition. Additionally, constraints can be attached to the state transitions. Behaviour diagrams are similar to Harel state charts [Har88].

4.2.5 Data Type Diagram

Data type diagrams represents user defined data types. Apart from predefined data types, TROLL allows the use of constructors for building complex data types.

4.3 Guidelines

Guidelines support developers during the software development process. We have already defined such steps in [Kow96, DH97]. Subsequently, we need various methods for an information system such as CATC to form a global method with a universal character. The questions that arise are: why do we need a new method? Can we not use an informal method together with the TROLL notations? Our experience with an informal object oriented method in this project was negative. The need for an abstract model to cover all aspects of the complex system CATC is not satisfied by informal methods. One of the main problems of the informal method is that they require the designer to reflect about possible implementation aspects. Our application domain and its data are too complex to mix design and implementation without loosing the global view of the system. Furthermore, our requirements were not supported by such an informal method. During the phase of modelling and implementation of our case study, we analysed the individual models and the case study was used to precisely find our own method. Then, we tried to find out in which way to model results in order to better understand the problem. For this process, empirical data was collected. Further, we had to combine different procedures in order to develop a method that could test our complex information system CATC according to correctness and efficiency. We derived the following guidelines from our experiences during the whole process of our case study. These can be summarised in the following steps:

- Finding objects and classes using object declaration diagrams
- Describing the static structure by means of community diagrams
- Representing interactions between objects using communication diagrams

Behavior Diagram

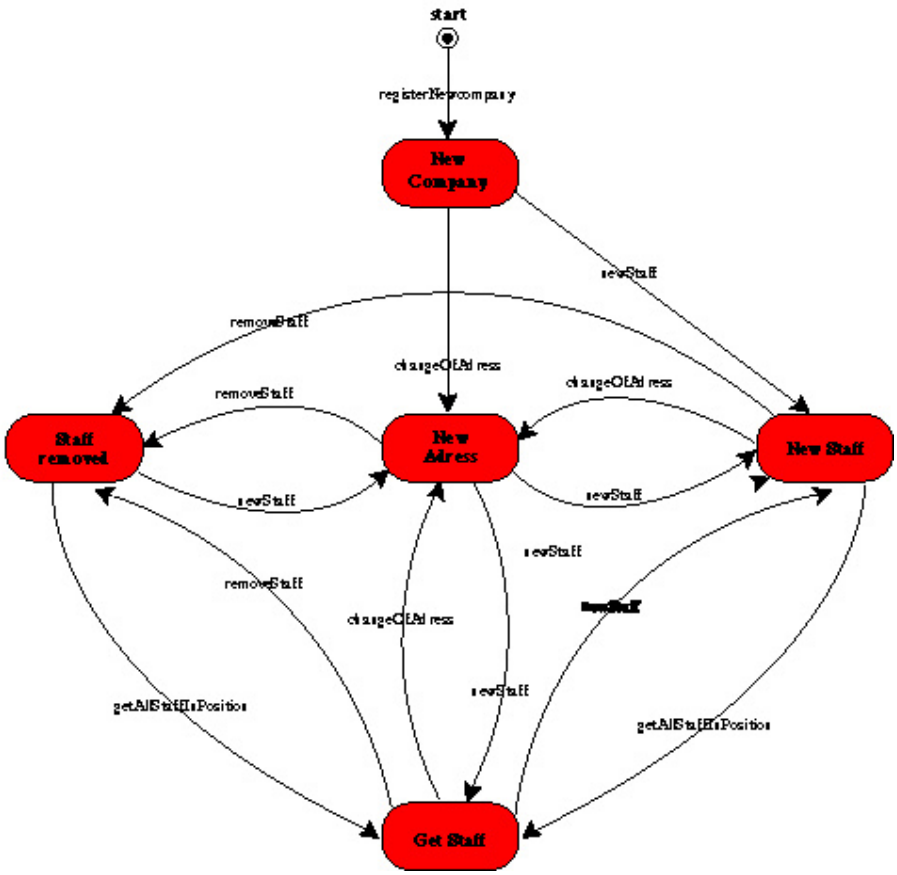


Figure 4.3: Behaviour Diagram

- Expressing the behaviour of the objects through behaviour diagrams
- Extending the object and data diagrams
- Transforming the specification into the relational model and C++.

In order to specify a system, the following steps can be set up to an overall process description:

Specifying a new System

- Defining the nodes of the system
  1. Hardware node
  2. Application node
  3. Domain node
- Describing the static structure of each node of the system
  1. Finding objects and classes
  2. Defining data types
- Defining global interactions between objects in different nodes as well as locally in the same node
  1. Declaration of attributes
  2. Definition of necessary actions
  3. Definition of birth and death actions
- Expressing the behaviour of the objects (the first time only graphically)
  1. Describing the hardware elements (trivial)
  2. Defining the business process (complex)
- Refining object declarations
- Validating the specification

What we are trying to emphasise is that

- Formal methods can be applied in a practical and adequate way
- TROLL may be a *part* of the development process of complex information systems.

As already mentioned in Chapter 2, software life cycles are an important part of the development process. Because TROLL is a formal object oriented specification language and is designed for the conceptual modelling or requirement specification phases, our guidelines just need to be understood for these phases, although we used TROLL successfully during the whole process.

The guidelines are defined as follows:

- Find out the concurrent objects (the nodes of the system). They "live" independently of each other and only have communication relationships.
- Describe the static structure for each node in the system (components and specialisations).
- Define the signature of the object classes (not only birth and death actions)
- Define user defined data types (simple and complex data types).
- Describe the global interactions and entailed local communications (direct calling and indirect calling).
- Extend the signature of the object classes.
- Express the local behaviour of each object class (life cycles, effects on attributes, constraints).
- Validate the specification.

Some hints for the correct application of the guidelines are:

- Distinguish between user interface node, information node, hardware node
- Make dialogue boxes as components of the user interface node
- Define two different actions for dialogue boxes:
  1. Close the dialogue boxes
  2. Delete the dialogue boxes
- Specifying behaviour diagrams for the hardware is not very useful
- Apply C++ and RDM transformation rules.

An important issue concerns the communication between object classes. Objects communicate via action calling. In TROLL action calling is directed and synchronous. We have to distinguish between two kinds of communication: (1) a communication between a complex object and its subobjects and (2) a communication between concurrent objects.

We will present our results by example. The case study is introduced in the next subsection. The specification of the case study using the TROLL guidelines is explained in Section 4.4

4.3.1 Running Example: Remote Controlling of Valves

VENTIL[1] is a programme to remote control and monitor explosion test stands in the test laboratories (see Figure 4.4). It is part of the experimental test software of CATC (see Figure 3.2).

[1]This is the German word for "valve"

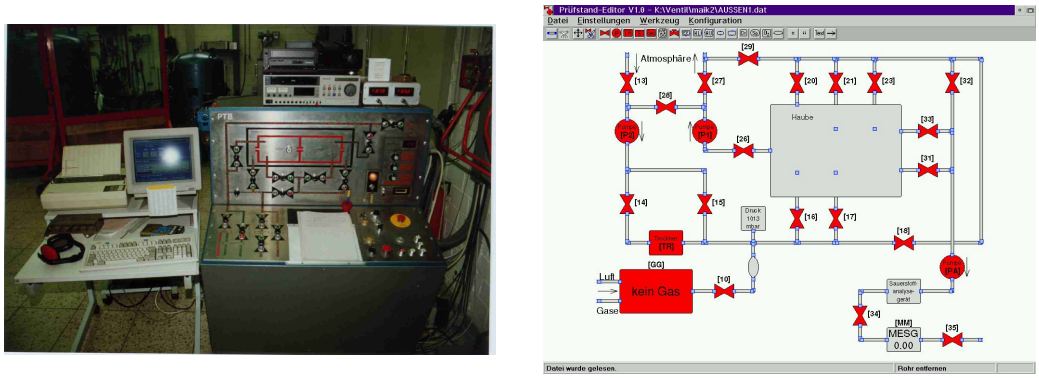


Figure 4.4: Hand Valve

The critical places for an explosion transition are the *joints*, i.e. the places where corresponding surfaces of two parts of an enclosure come together and result in a *gap*. For a flame-proof enclosure, each gap must be narrow enough so that only *flame-proof joints* are formed [ORW83].

The main equipment of an explosion test stand contains a gas source, the autoclave, analysis tools, pumps and valves. All of these devices are connected in a net of tubes and pipes. Figure 4.5 shows a schematic view of the smallest explosion test stand in the test laboratory, the so-called *Ex–Eva*[2]. VENTIL is used to control and monitor most of the devices of the Ex-Eva in order to create an explosive atmosphere in the autoclave. The actual measuring of explosion pressure during an experiment is done separately [Hoh96, Sha96].

Aside from the obvious tasks of allowing users, i.e. *testers*, to mix gases, open, close, and monitor valves, turn on and off pumps etc., VENTIL provides two more advanced features: the automatic *observance of dependencies* between devices and the *calculation of the gas flow*.

Observance of Dependencies

VENTIL prevents operators from accidentally violating *dependencies* between devices. Dependencies are rules which must be observed to protect the equipment and the environment (including the operators themselves) from the explosion test stand. The dependencies can be formulated as a kind of "master-slave" function where one device depends on the state or state change of one or more other device(s).

The following dependencies (to which we will refer throughout Subsection 4.4.2) are needed to protect the fragile oxygen analyser of the Ex-Eva (cf. Figure 4.5) from extreme pressure and soot developed in the autoclave during an explosion:

1. Valve 31 may be open if and only if valve 26 is open
2. Before valve 31 is opened, valve 34 is opened automatically
3. Before valve 34 is opened, valve 35 is opened automatically
4. One second after valve 31 has been closed, valve 34 is closed automatically

[2]**E**xplosions–**V**ersuchs **A**nlage (German for "explosion test stand")

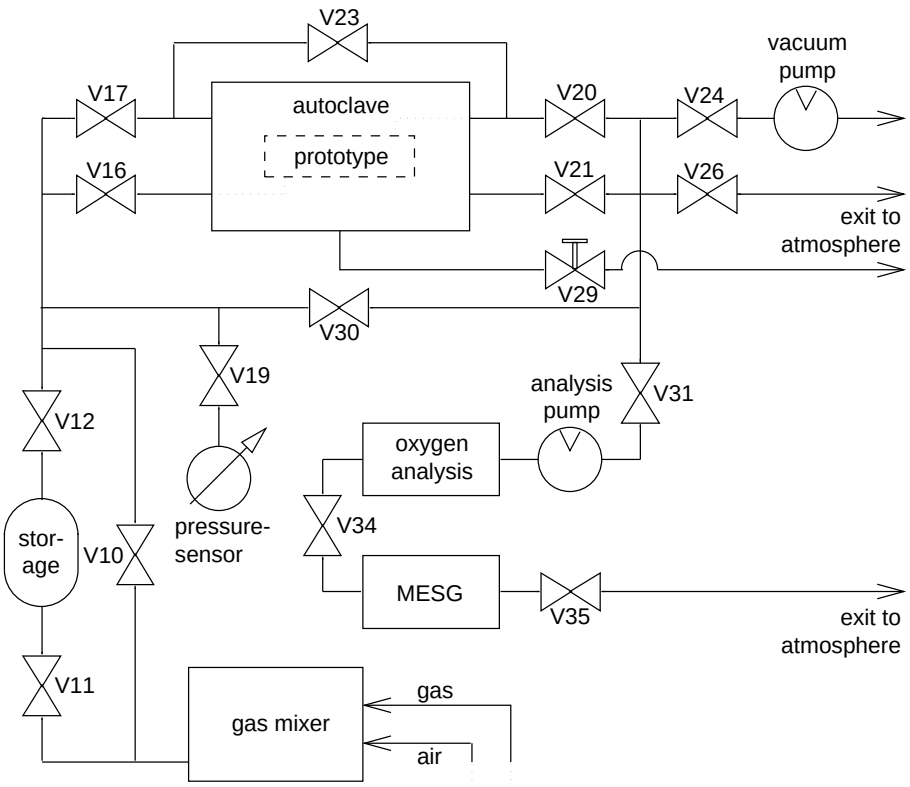


Figure 4.5: Schematic View of the Ex-Eva. Vxx denotes a valve.

5. After valve 34 has been closed, valve 35 is closed automatically.

Calculation of the Gas flow

In a schematic display similar to the one of the Ex-Eva in Figure 4.5, VENTIL offers the testers a calculated view of the gas flow in the test stand. This calculation is very important and depends on parameters like the expected - but not yet measured - gas pressure and the pumping direction of the pumps. We will not go into the details of the parameters and will instead use significantly simplified requirements for the visualisation of the gas flow here:

- A gas flow may begin or end at any gas entry or exit point of the test stand, e.g. exit into the atmosphere, into the gas mixer, into the reservoir, into the autoclave or into the pressure sensor. This is generally referred to as *endpoints*.

- There must be an "open way" from one endpoint to another, i.e. all valves need to be open and pumps turned on during gas flow. All other devices such as the oxygen analyser do not influence the gas flow and are therefore simply treated as pipes here.

Nevertheless, these reduced requirements will still be sufficient to present the implications of the gas flow calculation as far as this thesis is concerned. An extended description is given in [Sch96a].

4.4 Specification of VENTIL using TROLL Guidelines

The main components of the TROLL specification of VENTIL are presented in this section. First, a general overview of the object hierarchy of the information system nodes is given. Afterwards, the two most interesting parts of the specification are treated in detail: the observance of dependencies and the calculation of the gas flow.

4.4.1 Overview

Defining the nodes of the system

The specification of the VENTIL system is made up of three *nodes* (see Figure 4.6), namely the user node, the hardware node and the information system node. The user node describes the possible behaviour of the different user groups (testers, technicians etc.) and their interface to the main system. For instance, in the `Tester`[3] the object class - which is a part of the user node - specifies that valves can be opened and closed and which data must be provided for the gas mixer. These specifications solely focus on functionality and data and are therefore abstractions of possible implementations (like dialogue boxes or other user interface elements). Digital outputs, e.g. "open valve", digital and analogue sensors ("valve is open", voltage representing measured pressure) etc. are modelled in the hardware node. One merit of specifying VENTILin TROLL is the possibility to examine the information system node which is isolated from the other nodes and describes user interaction [Sch96a] and hardware behaviour [Hoh96].

In this thesis, the hardware and user nodes and their global interactions will not be treated any further. We will only discuss the specification of the information system node, beginning with the introduction of its object classes in the remainder of this subsection.

Describing the static structure of each node of the system

The Community Diagram in Figure 4.7 gives an overview of the component and inheritance hierarchies used in the information system node of VENTIL.

The Object Class `Knot`

The calculation of the gas flow requires the most complex algorithm in VENTIL. Hence, the structure of the specification was designed to suit this algorithm best. ¿From Figure 4.5 and its description, it is obvious that gas flow can be formalised as a path in a *directed graph* representing the explosion test stand. The *nodes* of the graph stand for the devices. Subsequently, devices also subsume the joints between two or more pipes and the entry and the exit points of the test stand, e.g. the external gas supply of the test stand and the *vertices* for its pipes[4]. Although pipes are generally undirected, the graph's vertices need to be directed here because gas

[3]throughout this thesis, we will print all terms referring to the TROLL specification in `typewriter` font and TROLL keywords in *italics*.

[4]To simplify reading, we will no longer distinguish tubes from pipes.

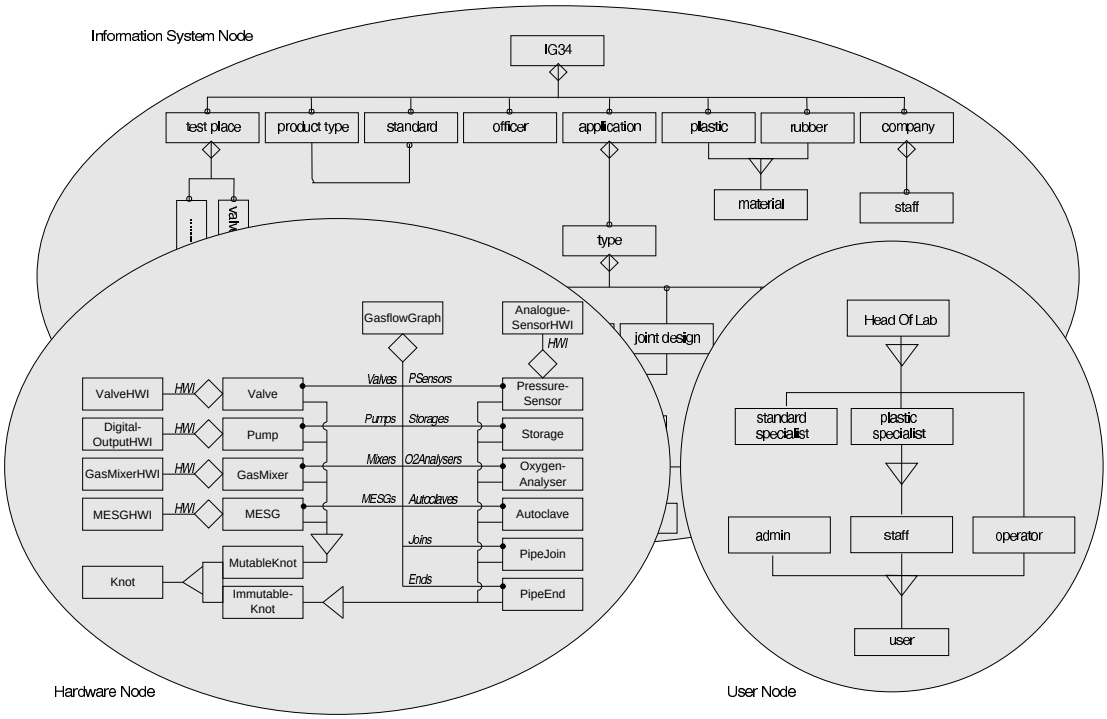


Figure 4.6: OMTroll Community Diagram of CATC. The triangles symbolise inheritance, the diamonds components.

can only flow one way, determined by the pumping directions of the pump and the gas mixer.

All basic properties of a node in the graph are modelled abstractly, i.e. instances of the object class `Knot` are not possible. It is a superclass of object classes representing a concrete device[5]. Hence, the devices become nodes of the graph, but do not need to take care of their connections to other nodes or their behaviour during the gas flow calculations themselves. Each device class inherits the basic properties of a `Knot`. Evolution within these properties does not have any effect on object class apart from `Knot`, thus facilitating the maintenance of the specification a great deal. All devices, including those which may be added to the test stand in future, reuse the specification of `Knot` and are therefore modelled more quickly and understandably. Furthermore, a `Knot` does not need to know to which types of devices it is connected because it does not need any specialised properties from its neighbour `Knot`s.

The Object Classes `MutableKnot` and `ImmutableKnot`

Devices like the oxygen analyser or pressure sensors can not be manipulated through VENTIL. With regard to their `Status` in the gas flow, those devices are always `open`. They are modelled as subclasses of the abstract object class

[5]While discussing the specification of VENTIL, we will use the name of a real-world object which is synonymous to its representing Troll object, e.g. by "valve 31", we generally mean "the object representing valve 31 in the specification". The few exceptions are made clear by phrases like "the hardware of valve 31".

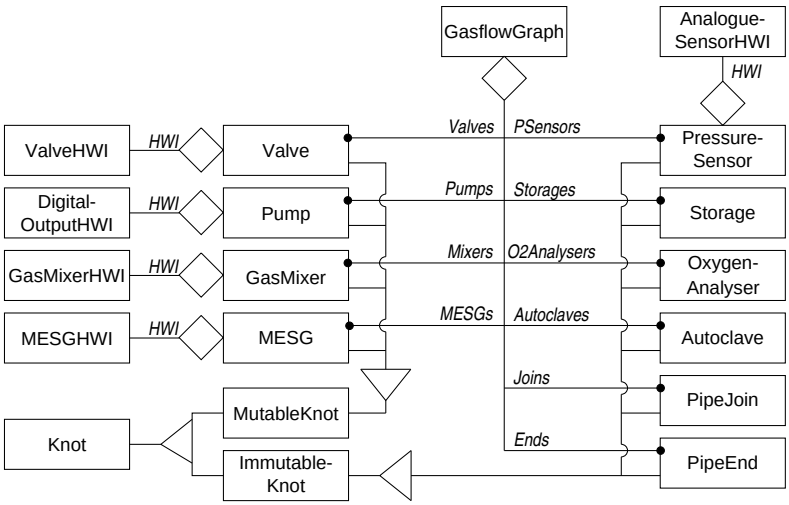


Figure 4.7: Community Diagram of the Information System Node of VENTIL.

`ImmutableKnot` which is a subclass of `Knot` (see Figure 4.7). `ImmutableKnot` constrains the `Status` to `open` (see the object behaviour diagram on Figure 4.8c) and disables the inherited switching operation. Devices that can be controlled by testers such as valves and pumps also have a common abstract superclass, `MutableKnot`. Obviously, only `MutableKnot` may need to observe dependencies because only if the state of advice is mutable, it can depend on the state of another device. Hence, the observance of dependencies is dealt with in `MutableKnot`.

The Device Object Classes

The different device classes of the test stand are modelled as separate object classes in VENTIL. Each of these object classes is a subclass of either `MutableKnot` or `ImmutableKnot` and hence indirectly a subclass of `Knot`. Several device object classes have components specifying hardware interfaces. By convention, the names of hardware interface classes end on `HWI`. For instance, `ValveHWI` models the interface to an object class within the hardware node of VENTIL. `ValveHWI` provides actions to open and close a valve and to check the current status of the hardware etc.[Hoh96].

Defining global interactions between objects in different nodes as well as locally through the same node.

Fulfilling Duties

The declarations of `MutableKnot`, as far as the observance of dependencies is concerned, look like this:

```
object class MutableKnot
aspect of Knot on ...                          -- Knot is the superclass of MutableKnot
attributes DutyList :  set(duty);
           DelayedDutyList :  set(delayedDuty);
           ...
```

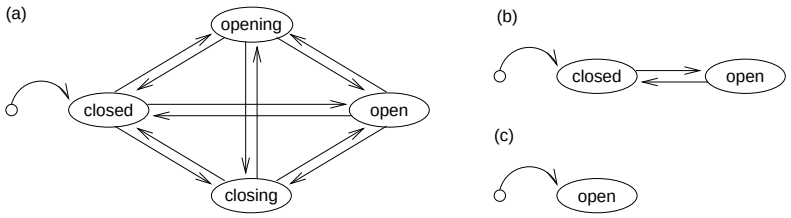


Figure 4.8: Object Behaviour Diagrams: (a) `Valve`, (b) `Pump`, (c) `ImmutableKnot`.

```
actions AreDutiesFulfilled(trigger:switch, ! now:bool, ! before:bool)
        FulfillDuty(duty:duty)
        FulfillAllDuties(trigger:action, exec:execution)
        FulfillDelayedDuties()
        Switch(action:switch, duties:set(duty))
        ...
end;
```

`AreDutiesFulfilled` returns (denoted by a '*!*') for a given action whether all `now` and `before` `dutys` in the `DutyList` are fulfilled. The return values are used by the switching operation of specialised `ImmutableKnots` to determine whether the desired `action` is allowed now or later or must be rejected. `FulfillAllDuties` calls `FulfillDuty` to fulfill all `dutys` in the `DutyList` for the given `trigger` and `exec` parameters, e.g. to fulfil all `dutys` `before` the `MutableKnot` is `activated`. Similar to `FulfillAllDuties`,`FulfillDelayedDuties` is used to process the `delayedDutys` in the `DelayedDutyList` as soon as their delay time has expired. The action `Switch` is inherited from the superclass `Knot` (see Subsection 4.4.1). Here, we introduce the second parameter, the set `duties`. All members of `duties` are added to the `DutyList`. Usually, `duties` is empty, but to fulfil a `before` `duty`, one new `duty` is passed.

Expressing the behaviour of the object (the first time only graphically).

Figure 4.8 shows the Object Behaviour Diagram of `Valve`; due to mechanical malfunction, any state transition is possible. The `Status` of a pump can only either be `open` (turned on) or `closed` (turned off) (see Figure 4.8b) and it is enforced by a constraint. The `Type` and `Status` attributes and the actions `FindFlowNo` and `FindFlow` are needed in the gas flow calculation.

The enumeration `switch` generalises the notions of "opening a valve", "turning on a pump" etc. to `activate` the respective counterparts to `deactivate`. It is used as the first parameter to the action `Switch` which is overloaded in any subclass of `Knot` to perform the required task for the individual subclass. Finally, the `Name` is a user-defined identification of a `Knot`. It is simply a three character string like "V11" for valve 11. For every operation if a user wants to work on a specific device, he inputs the `Name` to denote the device.

Refining object declarations

The Object Class `Gas flowGraph`

The management of our graph and the initiation of the gas flow calculation are modelled in the object class `Gas flowGraph`. The part of its specification relevant to this thesis is defined as follows:

```
object class Gas flowGraph
components Valves:  map (names) to (|Valve|);
           Pumps:  map (names) to (|Pump|);
           ...
attributes Knots:  map (names) to (|Knot|) derived
           Knots(name):= select knot from knot in dom(Valve)+dom(Pump)+...
                         where knot.Name = name;
actions Gas flow();
       Gas flowNo(no:nat, knots:list(|Knot|), flow:set(|Knot|),
                  ! newFlow:set(|Knot|))
   ;
constraints all name in names (
             cnt(select knot from knot in dom(Valves)+dom(Pumps)+...
                 where knot.Name = name) <= 1);
end;
```

In the *components* section, parameterised components are declared for each of the device subclasses of `Knot`. The parameter domains are always the range of possible `names` for `Knots`. In TROLL, it is necessary to specify the exact class of a component and not just one of its superclasses. It is therefore not possible to have one parameterised component containing instances of any of the subclasses of `Knot`. But since all `names` within a test stand are supposed to be unique even for different device classes, a well-defined map from `names` to `Knots` is required. This is achieved through the constraint given above. It states that each `name` may appear at most once in the union of all domains, i.e. the current existing instances of the parameterised components. Convenient access to the map from `names` to `Knots` is provided by the derived attribute `Knots`. The two actions `Gas flow` and `Gas flowNo` initiate the search for gas flows in the graph.

4.4.2 Observance of Dependencies

Classification of Dependencies

The formalisation of dependencies leads to a distinction between three classifications: static, dynamic and delayed dependencies[6].

Static Dependencies involve at most *one state change* in one device. This state change depends on the state of another device which is only observed, but not changed. Example 1 is a static dependency:

Example 1 Valve 31 may only be open - if and only if - valve 26 is open.

[6]Following the vocabulary of the engineers in laboratory 3.41, there is also a fourth type of "dependency" in the original requirement analysis. But its formal definition revealed that it must be treated differently from the other three [Sch96a].

Dynamic Dependencies always involve the possibility of *two state changes* in two devices, *as fast as possible*. From the point of view of one of the involved devices, there are three possible ways of executing their own change of state: *before* or *after* the other device or both in *parallel*. Specifying the parallel and after cases is straightforward. For the *before* case, a look will be taken at Example 2:

Example 2 Valve 35 must be opened *before* valve 34. The two following TROLL events must take place if valve 34 is commanded to open itself:

1. If valve 35 is already open, valve 34 opens and nothing else needs to be done. Otherwise, valve 34 commands valve 35 to open.

2. As soon as the hardware of valve 35 is opened, its corresponding object is notified and commands valve 34 to open.

Delayed Dependencies are special cases of dynamic dependencies: they involve the possibility of *two state changes* in two devices, but introduce a *delay time* between the switching operations. Obviously, *parallel* delayed dependencies do not make sense, thus leaving the *before* and *after* cases.

Delayed dependencies are treated similarly to other dynamic dependencies, but another event must be added. Example 3 shows a delayed dependency:

Example 3 Valve 34 must be closed one second after valve 31 is closed. Listing the required TROLL events for the closing command on valve 31, we get:

1. Valve 31 closes.

2. As soon as the hardware of valve 31 is closed, its corresponding object is notified. If valve 34 is already closed, nothing else needs to be done. Otherwise, the delay time begins.

3. As soon as the delay time has expired, valve 31 commands valve 34 to close.

Modelling Dependencies with Duties

The observance of any type of dependency is modelled in a system of *duties*. One dependency can result in a number of duties imposed on several devices (see Example 1). Duties are specified as *record* types in VENTIL. They are stored as attributes in the *duty list*[7] of the `MutableKnot`s they are imposed on. The duty list is checked before any switching operation is applied to the device. A duty *object class* is not helpful because all actions which process duties only modify attributes of `Knot`, but never the values of a duty (except for creation and deletion).

Duty types are modelled as follows in TROLL:

```
data type executionenum(now, before, parallel, after)
data type dutyrecord(trigger :  switch,
                     exec :  execution,
                     delay :  time,
                     target :  |Knot|,
                     action :  switch,
                     once :  bool)
```

[7]The name "duty *list*" emerges during development, although no sequencing is needed. See the declarations for `MutableKnot` below.

data type **delayedDuty***record*(`time` : *time*,
`duty` : `duty`)

The enumeration `execution` is used to distinguish static (`now`) from dynamic `duty`s. In the latter case, the time of execution of the second state change is given as either `before`, `after` or `parallel` to the first state change as explained above. The first component of the `duty` *record* holds the information on which `switch` the `duty` must be fulfilled, e.g. a `duty` with the `trigger` value `activate` imposed on a valve must be fulfilled each time the valve is opened. The `exec` component determines the type of `duty`. For `before` and `after` duties, `delay` holds the time between the first and the second switching operation; a delayed dependency has a value greater than 0. To fulfil the `duty`, `action` has to be passed to the `switch` operation of the `target`. The flag `once` set for `duty`s must be removed from the duty list as soon as they are fulfilled. A `delayedDuty` is an ordinary `duty` which must be fulfilled at a certain system `time`.

Lead by the examples introduced earlier, we will now take a look at how these actions work together and whether static, dynamic or delayed dependencies must be fulfilled. Fulfilling a static dependency is as simple as expected. Example 1 requires two `duty`s:

(`activate`, `now`, `0`, *Valve 26*, `activate`, *false*)

imposed on `Valve` 31 and

(`deactivate`, `now`, `0`, *Valve 31*, `deactivate`, *false*)

imposed on `Valve` 26.

While opening, the first `duty` must be fulfilled for `Valve` 31. The `Switch` action of `Valve` 31 checks the `Status` of the `duty`s `target`, *Valve 26*[8]. If `Valve` 26 is `activated`, i.e. the `Status` of `Valve` 26 is not `closed`, the hardware of valve 31 can be `activated` too. Otherwise, the switching command is rejected. Similarly, `Valve` 26 must check the status of `Valve` 31 before closing (according to the second `duty` given above).

The dynamic dependency of Example 2 results in the `duty`

(`activate`, `before`, `0`, *Valve 35*, `deactivate`, *false*)

imposed on `Valve` 26. What happens if the dynamic dependency of Example 2 must be fulfilled as shown in the Object Communication Diagram on Figure 4.9?

In the first TROLL event (continuous arrows), the `Switch` action is called to open `Valve` 34. `Switch` uses `AreDutiesFulfilled` to find out that there is at least one unfulfilled `before` `duty` and calls `FulfillAllDuties` with the parameters `activate` and `before`. `FulfillAllDuties` calls `FulfillDuty` to fulfil all necessary `duty`s, including the one of our example above. To fulfil the `duty`, `Switch` is called for `Valve` 35. The arguments passed are `activate` to open the valve (this is done by a call to the hardware interface object) and a *set*(`duty`) containing

[8]This is meant to be the identity of the object representing valve 26.

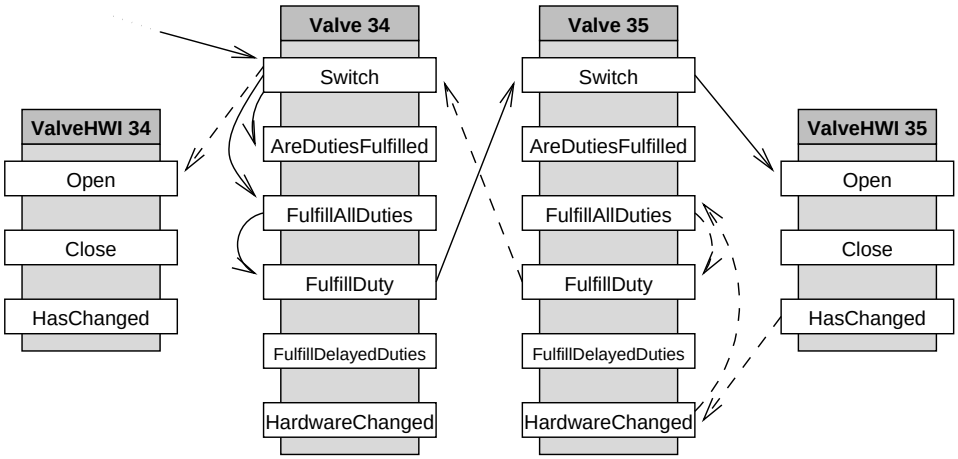


Figure 4.9: Object Communication Diagram for Example 2 (dynamic dependency).

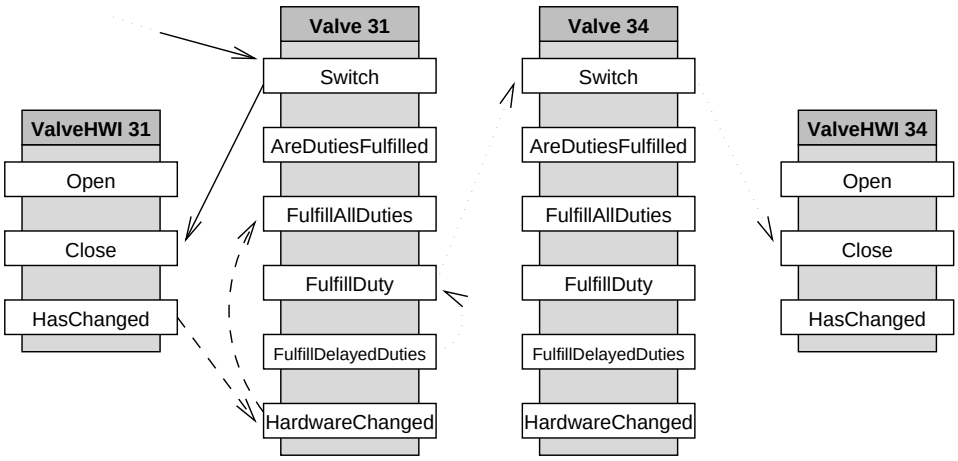


Figure 4.10: Object Communication Diagram for Example 3 (delayed dependency).

(activate, after, 0, *Valve 34*, activate, *true*)

This new duty is added to the DutyList of Valve 35. As soon as the hardware of Valve 35 is opened, the second TROLL event is initiated (dashed arrows). Because the event takes place after Valve 35 has been activated, the new duty must be fulfilled and then deleted from the DutyList (the last component of the duty is *true*). Fulfilling the duty results in the opening of Valve 34, thus we have Valve 34 opened after Valve 35 as required by the dependency.

Finally, we take a glance at Example 3 and the respective Object Communication Diagram (Figure 4.10). The duty which is imposed on Valve 31 is

(deactivate, after, 1, *Valve 34*, deactivate, *false*)

If the duty must be fulfilled (event one, continuous arrows), the hardware of Valve 31 can be Closed immediately because we speak of an after duty. After Valve 31 is closed, it tries to FulfillAllDuties (event two, dashed arrows). It therefore delays our example duty by adding

(current time + 1 sec),(deactivate, after, 1, *Valve 34*, deactivate, *false*)

to the DelayedDutyList of Valve 31. As soon as the specified time is reached, the third event takes place (dotted arrows): FulfillDelayedDuties calls once again FulfillDuty, and Valve 34 is Closed, too.

Here is a part of the TROLL specification of Knot:

```
data type vertexrecord(knot:|Knot|, flow:bool)
data type namesstring(3)
data type switchenum(activate, deactivate)
object class Knot
attributes Vertices:  set(vertex) isConstant;
         Type:  enum(endpoint, through) isConstant;
         Status:  enum(closed, opening, open, closing);
         Name:  names;
actions FindFlow(visited:set(|Knot|), flow:set(|Knot|),
                 ! newFlow:set(|Knot|), ! success:bool);
       FindFlowNo(no:nat, vertices:list(vertex), visited:set(|Knot|),
                  flow:set(|Knot|), ! newFlow:set(|Knot|), ! success:bool);
       Switch(action:switch, ...)        -- for the second parameter, see Subsect. 4.4.1
constraints cnt(Vertices) > 0,
           cnt(Vertices) = cnt(toSet(select v.knot from v in Vertices)),
           all vert in Vertices (vert.knot # self);
end;
```

A vertex object class is not used in the specification of VENTIL. It is sufficient to keep a set of references to neighbour Knots (together with a flag denoting whether this vertex is in the gas flow or not) to store outgoing vertices[9]. The three constraints on Vertices make sure for each Knot (i) that it is connected to at least another one (ii) that there are no two vertices to the same Knot and (iii) that there is no vertex to itself ((i) to (iii) are always fulfilled for an explosion test stand). Subclasses of Knot add constraints according to their specialised needs: a pump, for example, must always have one incoming and one outgoing vertex to denote the pumping direction. Note how easily allowed states of an object can be defined in TROLL. Constraint (ii) also serves as one of many examples in VENTIL where the expressive power of the *select* statement is exploited to yield a compact specification.

The constant attribute Type specifies whether a Knot is an endpoint of a gas flow or the flow just runs through the Knot. The Status attribute stands for the states the different devices may have. For a valve, open means it is open, opening that it is no longer closed, but not yet open (due to the mechanical switching delay), etc.

4.5 TROLL Workbench

The TROLL workbench is a set of software tools which supports the modelling and validation of TROLL specifications. The architecture and aims of the workbench was described in [Gra01] among others. When we began with our work, TROLL

[9]Specifying vertices is more complicated with the complete gas flow algorithm (using gas pressure confers to Subsection 4.3.1) because the graph needs to be traversed along incoming vertices as well. A direction part is added to the vertex data type and an additional constraint is needed to control the resulting redundancy [Sch96a].

workbench had not been fully developed, so that it could not be applied to its full extent in this project. Here, we only want to give a brief description of its functionalities. The workbench includes the following tools:

- *Workbench Management Tool*:
  This tool is the graphical front-end of the workbench and provides a common interface to the other TROLL tools.

- OMTROLL *Editor*:
  This editor provides specifiers with different diagrams for modelling the system in OMTROLL *(Community, Object, Communication and Data Type Declaration Diagrams)*. Figure 4.11 shows the specification of a

  part of the CATC system using the OMTROLL editor.

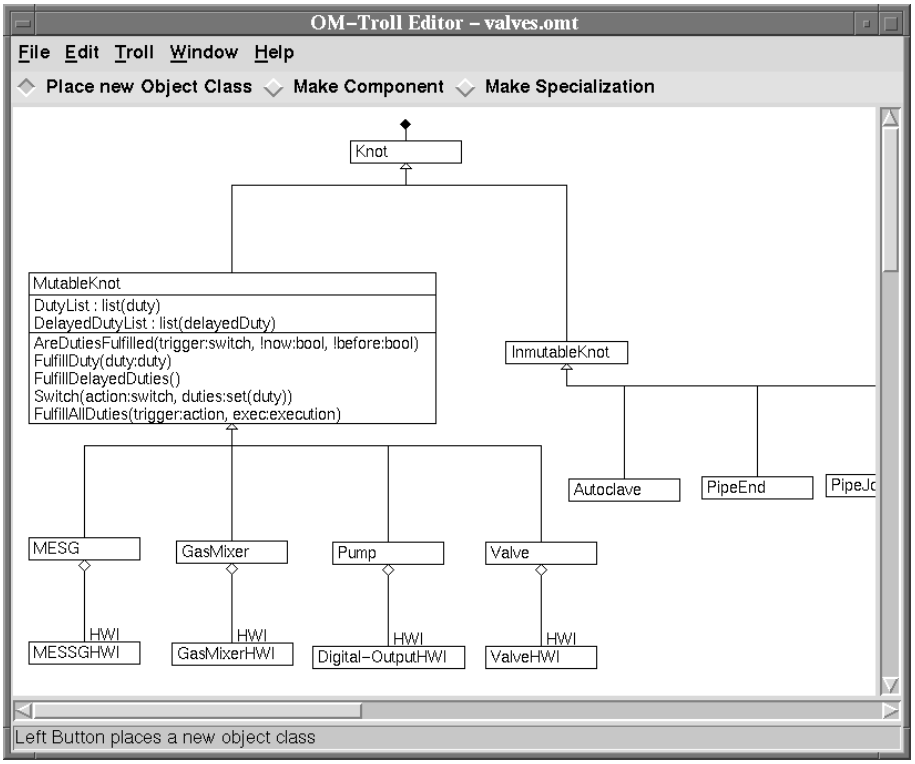


Figure 4.11: OMTROLL Editor

- TROLL *Editor*:
  This is a TROLL language mode for the (X)Emacs editors. Some of the new functionalities added to these editors are searching for specification components through the project files, automatic indentation and different colours and font styles for reserved words.

- TROLL *Checker*:
  This tool checks the syntax and static semantics of TROLL specifications. It can be embedded into the textual editor.

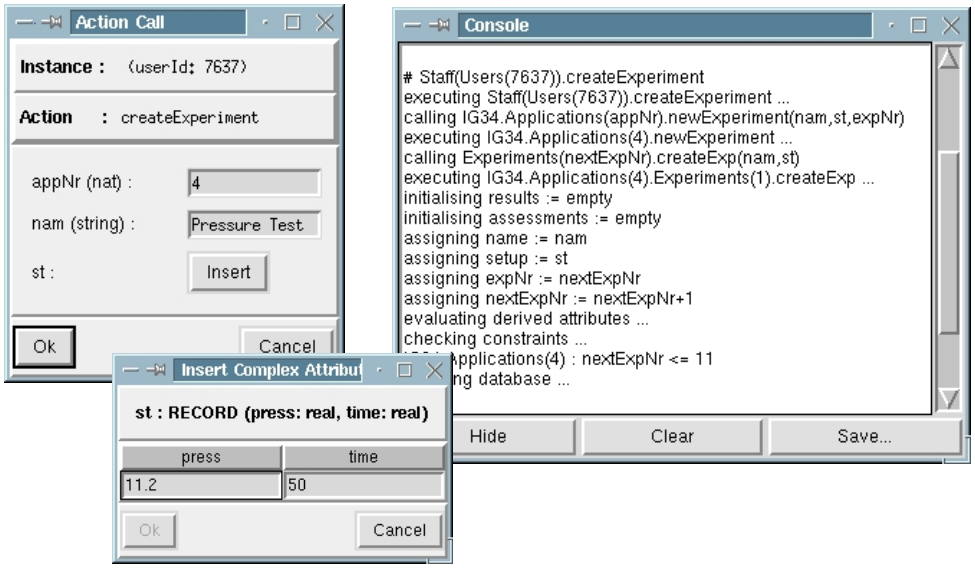


Figure 4.12: TROLL Animator

- TROLL *Documentation Tool*
  The documentation tool generates HTML code from TROLL specifications. The HTML code can be viewed in any HTML browser and allows users hypertext navigation through the specification and the introduction of informal comments.

- TROLL *Animator*:
  For the animation of the specifications, a database and a C++ library are generated to hold the object states and implement the model respectively. In the graphical front end of the animator users can create objects, navigate through their interfaces and simulate occurrences of events in the system to validate the specification against the user requirements. Figure 4.12 shows an animation session.

Chapter 5

Case Study Elaboration

In this chapter, we discuss the use of our method in the CATC development process, the experience gained, as well as the metrics applied to both the specification and the implementation.

5.1 The Development Process

The development process for this project began with an analysis phase for Group 3.4 in the PTB and went through design and implementation for this laboratory. In addition, we carried out specifications for other laboratories of this group. At that time, we worked on new features in our specification language, defined guidelines for our approach and evaluated them.

5.1.1 The Team

When the project began in 1994, the team consisted of 8-12 students and three full-time employees. All team members had similar backgrounds (computer science, mathematics) and therefore used the same terminology. One employee settled in the Federal Board had a computer science background as well as know-how in the problem domain. The two other employees were settled at the university with special interests in formal methods and mathematics. None of the students had any TROLL experience and know-how. The students, therefore, were trained in a special TROLL seminar which took place every two weeks. Both employees at the university were TROLL specialists and one of them was the designer of additional TROLL concepts and he spent much time answering specific questions. The students did not only have to learn to implement tasks, they were also involved in modelling the system. In the course of several meetings, all team members contributed in developing a first rough global concept. This complex task was then divided into smaller groups consisting of 1-2 students.

5.1.2 The Life Cycle Model

As mentioned in Chapter 2, one of the most important factors in a software development process is defining the life cycle model. Our experience showed that the

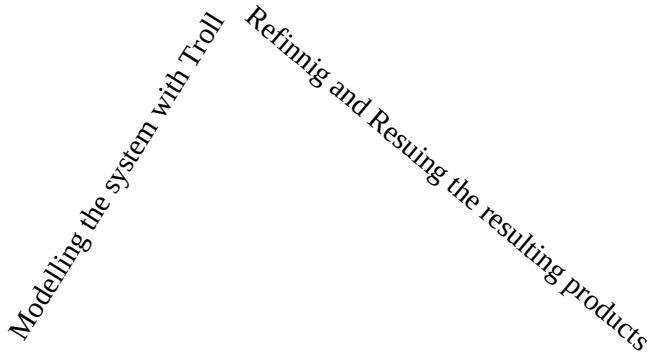


Figure 5.1: Life Cycle Model

fractal model is the best model to describe our process (see Figure 5.1). The whole process can be described as follows:

- modelling the Universe of Discourse with OMTroll
- realizing the object classes with Troll
- refining and reusing the resulting products with both Troll and OMTroll.

In the modelling part, a highly iterative, non-sequential process was followed by all students to widen their expertise in their sub-application domain (see Figure 5.2). We integrated and improved these steps later on in our method (see Chapter 4, Section 4.3) These steps are defined as follows:

Declaration of a concurrent system :

One main design decision is when a degree of concurrency is established. In this step, objects are either aggregated to complex objects or remain concurrent to other objects.

Description of objects of the system :

Aside from data types, objects systems had to be identified. To begin with, the structure and the behaviour of a set of objects had to be determined. Attributes and components were specified, actions were fixed and their effects on attributes were defined. Constraints on the behaviour of complex objects were modelled. This also includes the specification of interaction constraints in aggregated objects.

Specification of global interactions :

Due to the degree of concurrency, global interactions had to be specified. It was necessary to have a full understanding of all business processes in order to appropriately model the work flow and the data-flow.

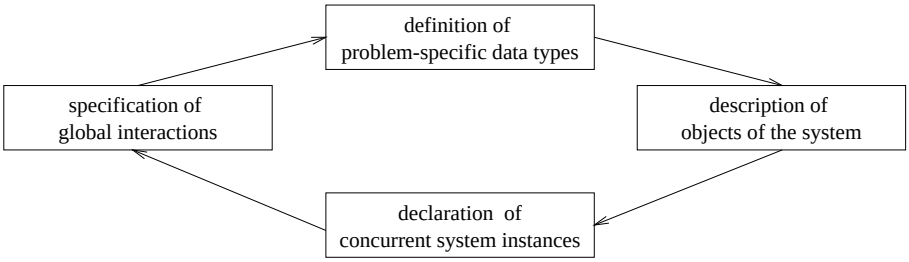


Figure 5.2: Development Process

Definition of problem-specific data types :

Many interviews with staff members or operators as well as investigation reports and formulas at the PTB had to be understood in detail for the data to be processed in the information system. This know-how had to be converted into user-defined data types.

5.2 Experience

5.2.1 Analysis

During the analysis phase, the focus was on providing a structured self-explanatory presentation of aspects that are relevant to the planned information system. This representation must be clear and precise, structured to support the management of the model complexity and independent of concrete implementations [JSHS96].

We specified the Universe of Discourse (UoD) of the problem domain in this phase rather than the application itself. One major problem in the modelling of the UoD was the identification of the relevant objects. The OMTroll community diagram was useful and self-explanatory enough to communicate with the users. During our first meeting, these diagrams had to be made understandable for the user. There were two types of users: the first user type had programming knowledge while the second user type had no such knowledge. The first user group tried to compare community diagrams with data-flow diagrams which resulted in a number of misunderstandings. The second user group tried to learn the concepts as well as possible which had a very positive result. Other diagrams made no sense to the staff and operators at PTB.

During our meetings, students tried to obtain an insight into specific sub-problem domains and all team members contributed in developing a first rough global picture. One advantage of using Troll was to reach a global picture before considering the details. Changes to the finer grained specification documents did not affect the global picture.

5.2.2 Design

The specification phase was an iterative process because misunderstandings came up only gradually. This made the re-specification of all parts necessary: user-defined

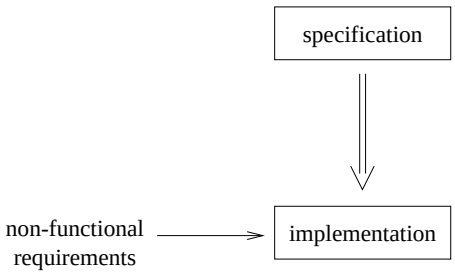


Figure 5.3: From Specification to Implementation

data types, specification of objects and changing the degree of concurrency together with the global interactions.

The students frequently communicated with us in order to clarify and refine interface definitions. Aside from bilateral discussions, we had regular meetings with all team members to propagate important design decisions concerning the interfaces.

The global architecture was not changed; it was only refined. The specification was therefore two-fold: first, we worked with general aspects using mainly diagrams and second we worked with more fine grain problems using textual notation. This involved two different aspects: a global view and a detailed view.

5.2.3 Implementation

One advantage of a formal specification is the independence of implementation aspects, although the implementation can be partly derived from the specification. As illustrated in Figure 5.3, the functional requirements which are part of the specification are translated into the implementation. It should be clear, however, that the non-functional requirements are not less important than the functional requirements. Non-functional requirements have been documented with functional requirements in a separate document [Kow94, HS94b]. The document is written in German and refers to a formal specification document when needed.

The separation of concerns led to a significant complexity reduction. In the early design phase, it was unnecessary to worry about non-functional requirements and constraints; the concentration laid on the functionalities of the system. A timing constraint on an application transaction, for example, at first has nothing to do with the business function the transition shall implement, i.e. its functional description can be separated from the timing constraint. The part of the specification which describes the information which must be stored together with the queries formalised in TROLL can be directly transformed into a database definition construct as usually done in the logical design step of a database development.

Before starting implementation, non-functional requirements had to be fixed. This had a major impact on the implementation phase, in regard to indexing the database, data replication, multi-process architecture with synchronous or asynchronous communication etc.

A relational database was used in our project. The relational and object oriented models are conceptually very different. Nevertheless, numerous solutions for mapping object concepts into relational tables have been proposed [Amb99, Kel97, BW95, RB97, Fus97]. To this end, we developed general transformation rules of TROLL specifications to relational database schemes [Bat96].

Not only structural aspects were modelled. There is much know-how in regard to how the system can evolve and behave because our approach is object oriented. This is reflected by translating the system into the target language of the application which in our case was C++. Information about attributes, actions and their effects on attributes, inheritance relations etc. were directly translated into the corresponding language constructs in C++. Once again, we developed general transformation rules. However, not only single TROLL concepts are translated; one can also take advantage of more complex information such as calling chains between several objects which were implemented using transactions. Thus, the functional requirements fixed by the formal model carried over into implementation and were enlarged by non-functional requirements.

5.2.4 Transformation from TROLL to C++

In this section various rules will be introduced which describe a transformation of a textual specification of TROLL into C++ code. These rules examine mainly the transformation of two of the most important concepts of TROLL: the data types and the object classes. For details see [Saa96, Sch96a].

Rules of Transformation of Data Types

TROLL Data Types	C++ Data Types
nat	short
int	int
real	float or double
string	char
union, enum	union, enum
union, enum	union,enum
record	struct
set, list, bag	will be replaced by OWL classes

Rules of Transformation of Object Classes

Each object class defined in the TROLL specification will be transformed into a C++ class. Transformation rules are shown in the following table:

TROLL **Object Class**	**C++ Class**
Class definition	Class definition
Attributes	Each attribute corresponds to a data element in the class definition. This data element has one of three characteristics: private, public and protected.
Actions	Each update action will be related to a method of the C++ class. The birth and death actions will be realised by the constructor and the destructor of the class.
Aggregation	By a indicator to the component class

In our project we distinguished between

- A user of a system as a physical person (see Figure 3.4)
- Data about a user stored in a system (see Figure 3.3).

The former specifies that the user's view of the system is taken into account, i. e. the functions a user of the system needs when interacting with it. It gives information about users along with the operation to change the information specified as an object class independent from the object class which represents the intended system. The latter is a component of the intended system (see [DH97]).

This distinction gave us the possibility to specify the user interfaces (dialog boxes) with TROLL. We used the *Object Windows Library* (OWL) which is a part of the Borland C++ Class Library, to implement our dialog boxes.

Rules to Translate TROLL Specifications within OWL Concepts

Rule 1: TROLL object class → a class derived from Tdialog

Different attributes are realised by an indicator for the OWL class TEdit. The data type string can simply be realised by this class, the data types nat, int and real must be realised by an additional control of the allowed input of the OWL class TFilter validator. This class checks the input while the user is still typing. The next rule is followed for the transformation of attributes with simple data types.

Rule 2: Attributes with Simple Data Types
TROLL string-attributes → TEdit class
TROLL nat, real, int-attributes → TEdit class+ plus TFiltervalidator class

The following shows the rule for the transformation of list valuated attributes and enumeration attributes by means of Object Windows Library classes:

Rule 3: Attributes with Complex Data Types
TROLL attributes: list attribute → TListBox or TcomboBox
TROLL enum attributes → TComboBox or TChekBox or TRadioButton

The list attributes were realised either by the class TListBox or the class TcomboBox, depending on the type of lists of this attribute. If the attribute is a variable list, where the user may cancel, adapt or search for applications, it will be realised by the class TBoxList. In the case that list valuated attributes realise a fixed list in which only one application may be chosen; the class TComboBox for the implementation of this attribute should be chosen. Attributes with an enumeration type enum may be realised either by the class TComboBox or TChekBox or TRadioButton.

Actions will be realised by method of classes. For the function of a user action such as "cancel", the allocation of a message by the response table is essential. The following shows the rule for the translation of TROLL actions into C++ functions.

Rule 4: TROLL actions → methods with entries of the response functions in the responsible table

The definition of an object class as a combination of several different classes will be realised by the concept aggregation in the TROLL specification language. Objects are created from this concept complex which possess different components. The aggregating object classes will be realised in C++ by classes called main classes. The relating components are declared as data elements within these main classes by means of an indicator to these classes.

Rule 5: TROLL aggregation → insert of indicators to the individual components as attributes of the main class

Example:

Figures 5.5 and 5.6 show a component declaration in TROLL and its transformation ino C++ respectively. The class TVgApplicationlist belongs to Basic Administration part of CATC (see Chapter 3). Via this dialogues, the staff can search for a certain application and get the necessary information about this application from the databases (see Figure 5.4). In order to extend the search, the TVgapplicationsearch is defined as a component of TVgApplicationlist.

5.2.5 Integration

Because the various CATC subsystems were not developed simultaneously, we began to integrate them using TROLL development environment TBENCH[Gra01]. We had very positive experiences in regard to the integration of various system parts which were modelled and implemented by different students. There were only a few problems in integrating the different modules because the students constrained themselves to the interfaces agreed upon with other team members[Goe97, Him97].

These problems can be mainly divided in two groups:

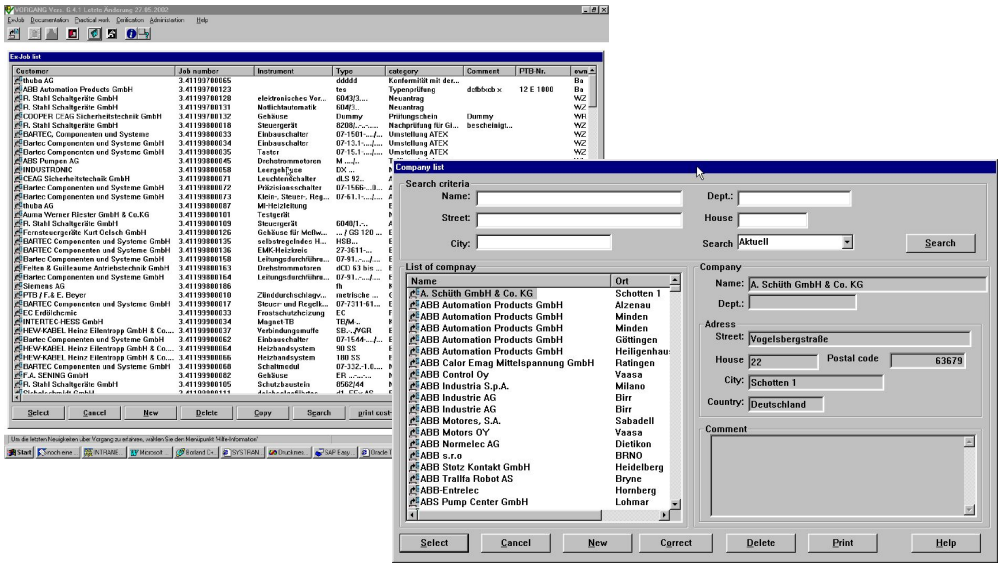


Figure 5.4: Application Dialog

object class `DlgApplicationlist ... end;`

Components `...`

`dlgsearch: DlgApplicationSearch;`

end.

Figure 5.5: Aggregation in TROLL

- Textual specification:
  - different declarations of the same attribute (with different types or different features as optional and constant etc.)
  - different declarations of the same data type (a data type declared as set and also as list)
  - violation of the TROLL restrictions regarding object visibility (the embedded directions for composition and specialisation are often confused.)
  - actions called with wrong arguments
- Inconsistencies between graphical and textual specifications.
  - changes in the graphical models were not always depicted in the textual ones and vice versa.

```
   class DlgApplicationSearch ... end;
   class TvgDlgApplicationlist: public TDialog
 ... end;
private:
TvgDlgApplicationSearch*  m_pDlgSearch;
....
(;
 class TvgDlgApplicationSearch: public TDialog
(
....
(; ...
   dlgsearch: DlgApplicationSearch;
   end.
```

Figure 5.6: Aggregation in C++

5.3 Learned Lessons

The formality and clearly defined semantics of the TROLL specifications carried over to the partly automatically derived implementations. Specification was the most important part of this project and this is reflected by the amount of time spent for modelling and implementation, respectively. On average 70% of our time was spent on specification purposes whereas only 30% of our time was spent on implementation, including integration.

During analysis and design, we spent more than 30% of our time manually checking syntax and semantic mistakes in our specification. This was not always an easy task with 17,000 lines of TROLL code. Here more tool support was necessary and re-doing already developed parts over and over again was disappointing. Tool support turned out to be one key factor to avoid frustration when changing a model again.

In [KKH+96], we anticipated only a few problems with the implementation because of minor programming errors. After completing the implementation phase, we knew that our forecast was correct, although we did not perform any formal checking or verification of the implementation against the specification. However, we carried out a careful code review. Codes and specifications were exchanged among the different team members. Everyone had to review the codes of another team member against the corresponding specifications. In this way, many errors were eliminated beforehand.

Approximately 400,000 lines of C++ code from about 17,000 lines of TROLL

have been written so far which is a ratio of 1:7. This shows that our input in earlier project phases was well worth the effort because it is now much easier to read a TROLL specification and understand a problem domain than to read the C++ code.

There have been a number of positive experiences by using the object oriented formal technique in our project.

- We developed a specification that successfully exploits the features that TROLL provides.
- The object oriented method of TROLL helped us to find a modular structure and well-defined module interfaces.
- Inheritance became an important factor in graph specification in VENTIL because we were able to separate general node properties (during gas flow calculations) from those of specialised device nodes. The general design and specification of a `Knot` is reused in every (future) device object class and no change in some device classes will influence other `Knot`s (see Chapter 4).
- Constraints that are often found in conjunction with the powerful descriptive *select* statement proved to be very helpful. These constraints allow for compact but simple restrictions of the possible behaviour of objects. Due to the similarities to the transaction concept of databases, the virtues of parallelism are obvious if a "usual" information system is designed. However, even in VENTIL concurrent execution has advantages, e.g. whenever one device is the target of several duties.
- The compact TROLL notation to C++ was surprisingly straightforward. The overall class structure to algorithmic detail in the gas flow calculations is an almost one-to-one relationship between specification and implementation on the design level. On the code level, however, the direct translation of the specification required to implement many additional classes to support TROLL data types (like sets of `Knot`s etc.) is more difficult.
- The object oriented method makes a distinction between formal and informal approaches. Informal object oriented approaches for problem analysis are relatively wide-spread by now. In general, these approaches mix informal entity based concepts for structure presentation with the data flow diagram for presentation of the system behaviour. An entire description of the structure and dynamics of the intended system, however, can not be reached.
- The SQL similar query part is used in TROLL to formulate standard queries.

  Furthermore, no continuous semantics exist in the above mentioned approaches. Semantics, however, are necessary requirements for verification of system characteristics. Our long-term goal is not only the specification of information systems, but to dispose of tools for animation, validation and verification.

- A TROLL specification refers to a goal system on a higher abstraction level which is independent of the tools used during later implementation. No exact description of the structure or implementation details are given which can occur at a later phase, e.g. on the basis of an increased output of the system.

- Definition of the substantial system functionality and user behaviour makes sense, especially for large systems. In general, the use of the formal method demands a high specification in the planning process. However, advantages such as early recognition of errors and simplified adaptability on altered requirements etc. have been observed. Furthermore, the CATC project showed that the specification efforts of the user resulted in high additional expenditures. Nonetheless, it delivers precious information that can be integrated into implementation.

The formality and clearly defined semantics of TROLL specifications were transferred into the implementation phase. In our project, we used a relational database and C++ as the target language for the application. General rules were laid down to transform structural aspects of TROLL specifications into relational database schemes and to translate information about attributes, actions and their effects on attributes, inheritance relations etc. directly into the corresponding language constructs in C++. On almost every design level, there was a one-two-one relationship between specification and implementation.

The advantages of specifying user interfaces and databases with a unified formalism are:

- The cooperation of both parts can also be modelled in this formalism so that any inconsistencies will be discovered at a design phases.

- We receive an implementation independent specification of the functionality of the user interfaces which may be implemented with the help of tools or programming languages.

- The result of the specification phase does not get lost, but is integrated into the implementation phase.

Before the use of TROLL, there was no common understanding of many models and numerous topics had to be discussed repeatedly when new people entered the project. There was a high personnel fluctuation since the students normally stayed in the project for only six months. Students that left the team took a lot of know-how with them and this was the information needed to give the models their semantics. This was a major reason to restart the project using the formal approach. It was much less critical with the formal approach when members left the project because the documentation they left was much less ambiguous.

Because different CATC subsystems were not developed at the same time, we started to integrate these using tools. Students had the possibility to share diagrams because tools were used for modelling CATC in other PTB laboratories. This helped in finding ambiguities by defining interfaces.

5.4 Application of Metrics

A large number of metrics have been suggested [LK94, Tha94] in literature to measure the quality of development process. A combination of several metrics is essential

because each individual metric measures only one characteristic. Presently, companies try to measure the quality of system design first because errors which remain undiscovered in this early phase can result in high costs later on in the implementation phase. To apply metrics manually, however, is difficult and costly. Tool support is necessary for syntax and semantic analysis of programme codes which must be measured. Another factor are the different quality characteristics. Some of these characteristics can be described as follows:

- reusability, effort to create reusable components
- extendibility, average productivity for code changing
- Ability to understand.

We decided to apply the following metrics for our project:

- NOA (number of attributes)
- NOM (number of methods)
- LOC (lines of code).

The LOC metric is easily defined and is the most often used metric. This factor, however, is not consistent from language to language, application to application and developer to developer. For this reason it is important to determine the steps precisely.

5.4.1 TROLL Specification

To make clear how the above metrics have been applied, this section presents the metrics for the Valve class from the TROLL specification. The metrics for its respective C++ class are presented later. Due to different LOC definitions, we have decided to count the number of lines of code which can be executed and the number of data definitions. Methods or attributes which are inherited at one point are not considered in the results, however, they may be added later on. The method *Switch* of the class *Valve* is an example for the application of LOC and how we counted the data definitions [Woz98].

```
Switch( Action, duty)
var                                          // Data Def.
   activ : bool,                             //     1
   now   : bool                              // +   1
                                             // ---------
                                             //     2

do                                           // Statements
  duty := duty + duty,                      //     1

  AreDutiesFulfilled(Action, activ, now),    //     1
```

```
  if (activ and not now) then                  //     1
    dutyFulfilled(Action, zp_before)           //     1
  fi,
  if (activ and now) then                      //     1
    if (Action = ac_makeactiv) then            //     1
      Hardware.On                              //     1
    else
      Hardware.OFF                              //     1
    fi,
    dutyFulfilled(Action, zp_parallel)          // +   1
  fi                                            // ---------
od;                                             //     7
                                               // = LOC(Switch)
```

The metrics for the class Valve are presented next.

```
/**
 **   File:            ventil.trl
 **   Author:          Martin Schoenhoff
 **   Transformation:  Maik Oppermann
 **   Version:         29.1.1998
 **/

object class Valve
aspect of MultipleKnot
  if Settingup(conf, idRef, duty) and conf.type = kt_valve

components
  Hardware : ValveHW hidden;

attributes                                              // Attribut
  IsDefect : bool                                      //    1
             derived (Hardware.Setup = vs_defect);

                                                  // NOA(Ventil) = 1
actions
  changing(new:valvesetup) hidden;

behavior
  Settingup(conf, idRef, duty) ...
  Start ...
  Stop ..
  Switch(Action, duty) ...      // We look at the switch
  MakeActive(info) ...
  changing(new) ...
  Hardware.changing(new) ...

constraints
  cnt(Neigbour) = 2,
  in = out;
end;
```

Method	LOC	Data Definition
Settingup	8	0
Start	2	0
Stop	2	2
Switch	9	0
MakeActive	1	0
Changing	7	0
Hardware.changing	1	0
NOM(Ventil) = 7	LOC(Ventil) = 30	2

The results for the entire VENTIL specification are as follows.

Classes are ordered in an alphabetical hierarchy. Indentations show inheritance.

Class	NOA	NOM	LOC	Data Definitions
Analogue sensor[1]	1	2	1	0
User[1,2]				
Administrator[1,2]	0	0	0	0
Tester[1,2]	0	0	0	0
Digital output[1]	1	3	2	0
Digital sensor[1]	1	2	1	0
Gas flowsgraph	7	12	68	16
GasmixerHW[1]	2	3	4	0
GMG_Dialog	5	8	12	1
Hilfe_Dialog	0	3	0	0
IG35[1]	0	2	2	0
Knots	16	13	27	15
Activeknot	7	8	29	10
Gasmixer	4	11	40	4
Handvalve	2	4	11	0
MESG	3	5	15	0
Pump	4	5	26	3
Dryer	3	5	24	2
Valve	1	7	30	2
Passiveknots	0	3	2	0
Berstingcontrol	0	1	3	0
Presssensor	1	3	7	0
Coveroutside	0	1	3	0
BoilerExEva	0	1	3	0
Boilershed	1	3	8	0
Pipeend	0	1	3	0
Pipeconnection	0	1	3	0
Oxygenanalyser	0	1	3	0
Storage	0	1	3	0
MESGHW[1]	2	3	21	0
MESG_Dialog	2	5	3	0
Measurement[1,2]	0	2	0	0
Testlab	0	2	4	0
PsAdmin	0	3	0	0
ValveHW[1]	2	4	3	0
Valvecontrol	2	18	66	1
ValveSystem	0	8	8	1
Vtservice	1	10	12	1
$\sum$	68	164	447	56

[1] only class specification exists, no C++ implementation

[2] As the classes are in different parts of the system, they were not taken into consideration.

5.4.2 C++ Implementation

As in the TROLL specification, the statements to be executed are counted without considering the complexity of the relevant statements in LOC. It would be desirable to have compiler support which can determine statements more precisely during syntax analysis and code generation. Due to different LOC definitions, we decided to count the LOC in addition to the number of data definitions applied. Special emphasis are placed on declarations following a constructor call because they are data definitions as well as executing statements.

The results of C++ implementation for the entire project will be presented in the appendix. Here, only a short example is given. For every class we consider declared attributes as well as methods. Inherited methods were not considered. However, they can be added on later.

Some classes of the project are inherited by the library class. The library class is not part of this project and for this reason, the measurement results for this class are missing.

C++ implementation consists of many classes in which the user interfaces are implemented (see examples of TvtVentilFrame). The number of classes in C++ is larger than in TROLL because some classes do not exist in TROLL, although they exist in C++. Moreover, TROLL data structures are translated by additional classes in C++.

Using the *Switch* method and the class *CvtValve* (Valve.cpp, Valve.hpp) as an example, we illustrate the application of the *NOA*, *NOM* and *LOC* metrics.

```
void CvtValve::Switch (const CvtAction Action,
                       const CvtDutyMounte& pmduty)
{                                                  // Statements  Data Def.
  CvtDuty pflNotNow                                      //             1
            = AredutyFullFilled                          //  1
              (Action, Cvttimeperiod::TP_Now);

  if (pflNotNow.GivetargetKnots() == NULL)               //             1
  {
    m_pmduty += pmduty;                                  //  1
    CvtDuty pflNotbefore                                 //             1
              = Aredutyfullfilled                        //  1
                (Action, Cvttimeperiod::TP_Before);

    if (pflNotbefore.GivetragetKnots() == NULL)          //  1
    {
      ostrstream strInfo;                                //             1
      strInfo << ...                                     //  1
      Information(strInfo.str());                        //  1

      if (Action == CvtAction(CvtAction::AC_Activate))   //  1
        m_pvthwHardware->On();                           //  1
      else
        m_pvthwHardware->Off();                          //  1

      dutyfullfilled (Action,
```

```
                          CvtTimePeriod::ZP_Parallel);   //  1
    }
    else
    {
      ostrstream strInfo;                                //  1
      strInfo << ...                                     //  1
      Information (strInfo.str());                       //  1

      dutyFullfilled (Action,
                          CvtTimeperiod::TP_Before);     //  1
    }
  }
  else
  {
    ostrstream strInfo;                                  //           + 1
    strInfo << ...                                       //   1
    Information (strInfo.str());                        // + 1
  }                                                      // ---       ---
                                                         //  16        5
}

//-------------------------------------------------------------//
//  V E N T I L . HPP                                           //
//  Project  : Valve Controllimg                                //
//  Notice   : Valve ACtivknots in Gas flowgraph.               //
//  Autho    : Martin Schoenhoff                               //
//  Date     : 24.10.95                                         //
//-------------------------------------------------------------//
class CvtValve : public CvtACtivKnoten
{
  // Membervariablen
  private:                              // simple Attribute
    ChwValve* m_pvthwHardware;         //             1

  // Memberfunction
  public:
    CvtValve (TvtGraphic* wGraphic,
              CvtShortform& kzshort, istream& isConfig);
    ~CvtValve();
    virtual void DetermineConnection (void);
    virtual BOOL AcceptDuty (const CvtDuty& Duty);
    virtual void FinishDrawing (TDC &dc) const;
    virtual BOOL IsDefect (void) const;
    virtual void Start (void);
    virtual void Stop  (void);
    virtual void Switch (const CvtAction Action,
                         const CvtDutyAmount& pmduty
                                        = CvtDutyAmount());
    virtual void StopSwitch (CvtAction Action);
    virtual BOOL MakeActive (void);
    void Changing (void);
```

};

Method	LOC	Data Definition	
		Constr.	Simple.
CvtValve	11	0	0
CvtValve	1	0	0
DetermineConnection	8	0	0
AcceptDutyn	2	0	0
FinishDrawing	32	0	2
IsDefect	1	0	0
Start	4	0	0
Stop	3	0	0
Switch	16	0	5
StopSwitch	1	0	0
Makeactive	2	0	0
Changing	22	0	2
NOM(Valve) = 12	LOC(Valve) = 102	0	9

The next table compares the results of the VENTIL specification and implementation.

Project	Attribute		NOM	LOC	Data Definitions	
Ventil	Obj.Valued	Simple			Constr.	simple
$\sum$ TROLL	-	68	164	447	-	56
$\sum$ C++	74	85	552	2709	95	276

Chapter 6

Summary and Further Work

6.1 Summary

This thesis has presented an approach and an evaluation of the object oriented specification language TROLL in PTB. Our cooperation with the PTB (Physikalisch-Technische Bundesanstalt - the German Federal Institute of Weights and Measures) began in 1994 [Kow94, HS94b] with the department responsible for explosion protection of electrical equipment working on a CATC project and ended in 2000. This department tests electrical equipment according to European standards for usability in explosive environments and issuing certificates for approved components. Only certified equipment can be used in hazardous environments like level meters in fuel tanks. The certification process consists of several steps:

- The applicant applies for certification of a certain component by the PTB
- The data of the application must be verified
- Technical specifications of the component have to conform to the relevant European standards
- If the application is correct, the specimen has to be checked whether it meets the specification or not
- If it does, the required tests have to be performed and documented. The certificate and other documentation is then issued
- If they do not, the reason for failure should be provided and the entire process must be repeated .

With an increasing number of applications (about 1000 certificates are issued per year),the requirement for computerised aid for the certification process came up. The main target of the CATC project was to develop an information system which supports the staff and operators in the department responsible for explosion protection of electrical equipment during different steps of the certification process. The system has to facilitate administrative tasks as well as experimental tests and design approval. The Computer Aided Testing and Certification System (CATC) therefore comprises modules such as :

- ADMIN which registers and administrates applications[Saa96, Bat96]
- Ex-Pert which determines and assigns experimental tests compulsory for a certain type of equipment as well as information retrieval on European standards used in design approval [Ben96, Jür96]

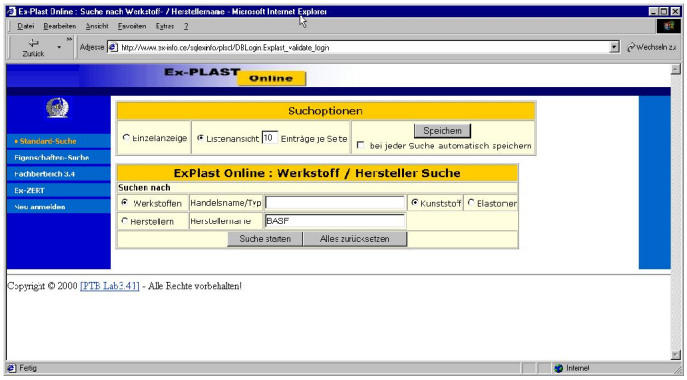


Figure 6.1: Explast via Internet

- Ex-Plast which retrieves information on synthetic material used in explosion proof equipment required in design approval [Sch96b]
- PRESSTEST which controls test equipment as well as automatic collection and storage of measuring values [Sha96, Hoh96]
- VENTIL which is used as a remote control of valves in experimental tests [Sch96a].

The activities realised by the project are often called business processes. The organisation complexity of the project as well as the system that was used to support this organisation was rather high. When the project started at the beginning of 1994, no formal object oriented methods were applied [Moh94, Arn95]. At the end of the year, it was clear that chances of success with the chosen approach were rather low. At that time, we decided to use TROLL from analysis to implementation. After a year, the following experiences were made:

- Our language was not suited for all aspects of our problem domain, for example in the area of real-time
- Troll helped us to find module structures and well-defined module interfaces
- The transition from the compact TROLL notation to C++ was straight-forward
- The question of guidelines became increasingly more important
- Tool support became crucial for the success of big projects.

Aside from its primary goal, the project serves as a field study for validating and developing the TROLL language and method for modelling and specifying information systems [DH97, GK97, Har97, HDK+97, GKK+98].

The project started in 1994 with an analysis and specification of a system for administration support. Full-time employees of the PTB split the work into separate tasks, supervised the design and implementation and integrated the results. Students graduating at this institute solved these tasks. This frequently included modelling the different parts of the systems under consideration with TROLL. This included database design and user interfaces as well as hardware and Internet/intranet access. The most recent developments (January 2000) were to allow Internet/intranet access to certain systems (Ex-Zert, Ex-Plast) see Figure 6.1 and adjusting the PRESSTEST subsystem to new hardware, porting it to a new operating system and making minor adaptations to its functionality.

We only concentrated on the guidelines of TROLL in this thesis. The development of Troll tools is done by [Gra01]. The workbench was used by students doing their master's theses in cooperation projects with the PTB. Several positive experiences have already been reported in [KG98, Kan99, Win00, Sch00].

The research in this thesis was motivated by the observation that although formal approaches to software modelling provide more precise specifications, there is a need for techniques to support methodology. Guidelines support developers during the design process and to some extent to determine possible steps. These steps can then be used for the development of the system. This is why we defined such guidelines for our specification language TROLL. This can be demonstrated in the following steps:

Specifying a new System

- Defining the nodes of the system
  1. Hardware node
  2. Application node
  3. Domain node
- Describing the static structure of each node of the system
  1. Finding objects and classes
  2. Defining data types
- Defining global interactions between objects in different nodes as well as locally in the same node
  1. Declaration of attributes
  2. Definition of necessary actions
  3. Definition of birth and death actions
- Expressing the behaviour of the objects (the first time only graphically)
  1. Describing the hardware elements (trivial)
  2. Defining the business process (complex)
- Refining object declarations
- Validating the specification.

We finished the modelling and implementation of the CATC system for one of three laboratories of the group responsible for explosion protection of electrical equipment and reused this model for other laboratories at the PTB. Without tool support, re-specification of the work and redoing of already developed parts was generally difficult and disappointing. Tool support was one key factor to avoid frustration when changing models. During analysis and design, we spent more than 30% of our time checking syntax and semantics mistakes manually in our specification. This was not an easy task with more than 17,000 lines of code.

The main part of this thesis describes the advantages and the disadvantages of using TROLL in various software engineering phases and different problem domains.

6.2 Future Work

The project will move from a purely national one to an international one.

The information system CATC has established itself within the PTB. Due to the positive experiences made with CATC, others departments have asked whether the system can be adapted to other situations. In the context of an international co-operation between the PTB and INSEMEX (Romania) ("(INSEMEX SECEEx Petrosani) National Institute of Mining Safety and Explosion Protection, Romania"), a modified and simplified English version of this information system has been introduced. Other requirements have already been made to the PTB. PTB are still working on an internationalisation of CATC.

The further development of CATC will find its application within the internet. The future task will be to establish an online expansion as an international and comprehensive gateway for explosion protection and safety.

The first step will be an online version of Administrationspart (ADMIN). Here, users can follow and carry out their tasks on a parallel basis. Co-operation between various partners will be intensified. This will make a faster and better execution possible.

A gateway for explosion protection could then include the following:

- A "Who is Who" register in which the different participants (testers, producers, operators and authorities) can introduce and describe themselves
- A training system with a calendar of events
- Discussion forums
- An extension of the existing offers (Ex-ZERT, Ex-PLAST, ADMIN)
- Search mechanisms for the information system
- Internationalisation of the system
- Different access rights for users.

The security plays a very important roll in such a system. An exact security analysis must be carried out. The modelling and implementation of safety in distributed systems must be precisely and accurately examined.

All of this will give us the opportunity to become more experienced in regard to reuse issues. Since most business processes and rules are formalised, we believe that we can easily adapt our models to this new dimension of the problem domain. This potential future may prove the strategic advantage of our choice of a formal approach. We had long-term discussions about the overall model and this lead to a good understanding of the general setting of the PTB world.

TROLL has also been used in other cooperation projects within PTB. After a year of successful work in the department responsible for explosion proof electrical equipment, we were asked by the department responsible for electricity whether we could work with them as well. This department concentrates on research and development in the field of electrical precision measuring techniques, especially with the representation, reproduction and dissemination of electric units in the area of semiconductor devices within the Department of Quantum Electronics. This department focuses on the possibilities offered by semiconductors, with the Quantum Hall Effect for example. The unit of resistance can be reproduced with a formerly unattainable precision and single electron tunnelling might be used likewise for current intensity. Measurements performed on pieces of semiconductor wafers, like photoluminescence excitation spectroscopy or single electron tunnelling measuring, produce a lot of data. The information system MDB2 (Measuring Data Database

of Division 2) was therefore designed and implemented to store this data and to provide access to it via a graphical user interface. TROLL was used for the design of both the database and the user interface. The first step in the development of the information system MDB2 was to design a database which should store data on specimens, chips and on the semiconductor wafers they were made from as well as on the experiments or measurements performed on them and the data obtained thereby [Wol98]. The next step was to complete the user interface

Another application of TROLL was to model the equipment used for the measuring and the set-up of these devices in different types of experiments aiming at the implementation of an information system that will facilitate work at different measuring places and provide computerised control over the measuring devices[Ara98]. While the MDB2 was designed to store data obtained by different types of measurements like single electron tunnelling or photoluminescence and photoluminescence excitation spectroscopy measuring, a second system was developed to retain information on the available equipment and the set-up of devices for the different measuring experiments as well as their executions. The first step here was to analyse the structures and proceedings of the different measurements and to extract the features which they have in common and which distinguish them - with a focus on control, input and output parameters and the sequence of actions performed. The result of this analysis was modelled using TROLL$^{\cdot}$
This specification could then be used as a guideline for developing homogenous modules for the different stages of experiments which in turn provide the basis for implementing programmes to facilitate the measurements (by means of computer-based control over measuring devices and data processing). As a second step, the specification was evaluated with the implementation of a system supporting the photoluminescence spectroscopy measurements.

Bibliography

[ABL96] JR. Abrial, E. Börger, and H. Langmack. *Formal Methods for Industrial Applications*. Springer, Germany, 1996.

[Amb99] S. W. Ambler. Mapping Objects to Relational Databases. White Paper, 1999. Available at htttp://www.ambysoft.com/mappingObjects.html.

[Ara98] S. Arabestani. Objektorientierte analyse und spezifikation der meßwerterfassung von tieftemperatur-hochfeld-experimenten. Diploma thesis, Technical University of Braunschweig, 1998.

[Arn95] U. Arnold. Entwurf eines informationssystems für kunststoffe explosionsgeschützter elektrischer betriebsmittel. Diploma thesis, Technical University of Braunschweig, 1995.

[BAM+02] R. Bourque, P.and Dupuis, A. Abran, J.W. Moore, L. Tripp, and S. Wolff. Fundamental principles of software engineering -a journey. *Journal of Systems and Software*, 62(1):59–70, 2002.

[Bat96] M. Bathik. Objektorientierte Realisierung der datenbankschnittstellen eines informationsystems zur verwaltung der prüfvorgängein der ptb. Diploma thesis, Technical University of Braunschweig, 1996.

[Bay99] Z. Bayram. Business Object-Oriented Analysis and Design (BOOAD) Methodology. *JOOP*, 12(1):59–67, 1999.

[Ben96] S. Bengsch. Objektorientierte modellierung und prototypische implementierung des informationssystems expert an der ptb– datenbankentwurf. Diploma thesis, Technical University of Braunschweig, 1996.

[BH96] A. Behforooz and FJ. Hudson. *Software Engineering Fundamentals*. Oxford University Press, Inc., Oxford, 1996.

[Bjo98] D. Bjorner98. Domains as Prerequisites for Requirements and Software. Domain Perspectives and Facets; Requirements. In *RTSE'97: Requirements Targeted Software and Systems Engineering*, volume 1526 of *LNCS*, pages 1–41. Springer, 1998.

[BP01] L. Baresi and M. Pezze. On Formalizing UML with High Level Petri Nets, in Concurrent Object-Oriented Programming and Petri Nets. *Eds. Lecture Notes In Computer Sciences*, 2001.

[BW95] K. Brown and G. Whitenack. Crossing Chasms, A Pattern Language for Object-RDBMS Integration. White Paper, Knowledge Systems Corp., 1995. Available at http://www.ksccary.com/articles.htm.

[CSS89] J.-F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 Users Manual. Internal Report.* INESC, Lisbon, 1989.

[DB95] P. Du Bois. *The Albert-II Language – On the specification and the Use of a Formal Specification Language for Requirements Analysis.* PhD thesis, Facultes Universitaires Notre-Dame de la Paix Namur, Belgien, 1995.

[DDBZ95] E. DuBois, P. Du Bois, and J.-M. Zeippen. Specifiying a Mobile Telephone Protocol with AlbertII. Technical report, Facultes Universitaires Notre-Dame de la Paix Namur, Belgien, 1995.

[DE95] G. Denker and H.-D. Ehrich. An Event-Based Semantics for Transactions. In G. Bernot and M. Aiguier, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability (IS-CORE'95), Technical Report, Evry, Sept. 1995*, pages 57–72, Bd des Coquibus, F-91025 Evry Cedex, France, 1995. Universite d'Evry Val d'Essonne, Laboratoire de Mathematiques et d'Informatique.

[DE97] G. Denker and H.-D. Ehrich. Specifying Distributed Information Systems: Fundamentals of an Object-Oriented Approach Using Distributed Temporal Logic. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), Volume 2, IFIP TC6 WG6.1 Intern. Workshop, 21-23 July, Canterbury, Kent, UK*, pages 89–104. Chapman & Hall, 1997.

[DH97] G. Denker and P. Hartel. TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics. Informatik-Bericht 97–03, Technische Universität Braunschweig, 1997.

[DHS94] E. Dubois, P. Hartel, and G. Saake, editors. *Formal Methods for Information System Dynamics, Workshop of the CAiSE'94 Conference, Utrecht, 1994.* Univ. of Twente, Technical Report, 1994.

[DK92] E.H. Dürr and J.v. Katwijk. VDM++, A formal specification language for object-oriented design. In *Proceedings of TOOLS7 (Technology of object-oriented languages and systems).* Prentice-Hall, 1992.

[EC00] H.-D. Ehrich and C. Caleiro. Specifying communication in distributed information systems. *Acta Informatica*, 36(Fasc. 8):591–616, 2000.

[Eck98] S. Eckstein. Towards a Module Concept for Object Oriented Specification Languages. In J. Bārzdiņš, editor, *Proc. 3rd Int. Baltic Workshop on Data Bases and Information Systems, Riga, Latvia, April 15-17*, volume 2, pages 180–188. Institute of Mathematics and Informatics, University of Latvia, Latvian Academic Library, 1998.

[Eck01] S. Eckstein. *Module für verteilte Objektsysteme — Konzepte zur Strukturierung und Wiederverwendung objektorientierter Spezifikationen*, volume 77 of *Reihe DISDBIS.* infix–Verlag, Sankt Augustin, 2001.

[ECSD98] H.-D. Ehrich, C. Caleiro, A Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT'93)*, pages 453–467. Springer, Berlin, LNCS 668, 1993.

[EG01] S. Einer and A. Grau. Integrating Petri Nets and TROLL in the Modeling of Engineering Systems. In *Proc. of the IEEE TC-ECBS and IFIP WG 10.1 Joint Workshop on Formal Specifications of Computer Based Systems (FSCBS'01) Washington DC*, pages 7–12, April 2001.

[EGS90] H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. deBakker, W.P. deRoever, and G. Rozenberg, editors, *Proc. REX/FOOL Workshop*, pages 203–228, Noordwijkerhood (NL), 1990. LNCS 489, Springer, Berlin.

[EH96] H.-D. Ehrich and P. Hartel. Temporal Specification of Information Systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering, Proc. Int. Workshop in Honor of C.S. Tang,Beijing, 14-15 August 1995*, pages 43–71. World Scientific, 1996.

[Ehr96] H.-D. Ehrich. Object Specification. Informatik-Bericht 96–07, TU Braunschweig, 1996.

[EJDS94] H.-D. Ehrich, R. Jungclaus, G. Denker, and A. Sernadas. Object-Oriented Design of Information Systems: Theoretical Foundations. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*, pages 201–218. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 1994.

[EN 87a] CELENEC: Europäische Norm EN 50014. Elektrische Betriebsmittel für explosionsgeschützte Bereiche, Allgemeine Bestimmungen. VDE–Verlag, Berlin, Offenbach, 1987.

[EN 87b] CELENEC: Europäische Norm EN 50018. Elektrische Betriebsmittel für explosionsgeschützte Bereiche, Druckfeste Kapselung "d". VDE–Verlag, Berlin, Offenbach, 1987.

[EP00] H.-D. Ehrich and Ralf Pinger. Checking object systems via multiple observers. In *International ICSC Congress on Intelligent Systems & Applications (ISA'2000)*, volume 1, pages 242–248. University of Wollongong, Australia, International Computer Science Convetions (ICSC), Canada, 2000.

[ES90] H.-D. Ehrich and A. Sernadas. Algebraic Implementation of Objects over Objects. In J. W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness"*, pages 239–266. LNCS 430, Springer, Berlin, 1990.

[ES95] H.-D. Ehrich and A. Sernadas. Local Specification of Distributed Families of Sequential Objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Types Specification, Proc. 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, S.Margherita, Italy, May/June 1994, Selected papers*, pages 219–235. Springer, Berlin, LNCS 906, 1995.

[Esp93] Espirito Santo Data Informatica, Lisbon. *OBLOG CASE V1.0 – The User's Guide*, 1993.

[ESS88] H.-D. Ehrich, A. Sernadas, and C. Sernadas. Abstract Object Types for Databases. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, pages 144–149, Bad Münster am Stein, 1988. LNCS 334, Springer, Berlin, 1988.

[FK97] M. Fowler and S. Kendall. *UML Distilled Applying the Standard Object Modeling Language*. Addison–Wesley, USA, 1 edition, 1997.

[FL98] J. Fitzgerald and P. Larsen. *Modelling Systems Practical Tools and Techniques in software Development*. Cambridge, Cambridge University, 1998.

[FME97] Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, 1997.

[Fus97] M. L. Fussell. Foundations of Object Relational Mapping. White Paper; ChiMu Corp., 1997. Available at http://www.chimu.com/publications/objectRelational/.

[GK96] S.J. Goldsack and S.J.H. Kent. *Formal Methods and Object Technology*. Springer, London, 1996.

[GKK+98] A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger, and H.-D. Ehrich. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. In T.W. Ling, S. Ram, and M.L. Lee, editors, *Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore*, pages 277–290. Springer, LNCS 1507, November 1998.

[Goe97] K. Goehnert. Object-Oriented Integration of Database Interfaces in CATC. Diploma thesis, Technical University of Braunschweig, 1997.

[Gra01] A. Grau. *Computer-Aided Validation of Formal Conceptual Models*. PhD thesis, Technical University Braunschweig, Germany, March 2001.

[GV96] M. Gogolla and H.-D. Vlachantoni8uEhrich. Object Specification. Informatik-Bericht 96-07, Technical University of Braunschweig, 1996.

[Hal96] A. Hall. Using formal methods to develop an atc information systems. *IEEE Software*, pages 66–76, 1996.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[Har97] P. Hartel. *Konzeptionelle Modellierung von Informationssystemen als verteilte Objektsysteme*. Reihe DISDBIS. infix-Verlag, Sankt Augustin, 1997.

[HB95] MG. Hinchey and JP. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, UK, 1995.

[HD98] P. Heymans and E. Dubois. Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Specifications. *Requirements Engineering Journal, Springer-Verlag*, 3:202–218, 1998.

[HDK+97] P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. Information systems modelling with TROLL formal methods at work. *Information Systems*, 22(2-3):79–99, 1997.

[Hei97] Maritta. Heisel. Improving software quality with formal methods: Methodology and machine support. Habilitation Thesis, Technische Universtit Berlin, 1997.

[Hes97] W. Hesse. From woon to eos: New development mewthods require a new software process model. In A. Smolyaninov and A. Shestialtynow, editors, *WOON'97: International Conference on OO Technology*, pages 88–101. St.Petersburg, 1997.

[Hey97] P. Heymans. Some Thoughts about the Animation of formal Specifications Written in the Albert II Language. In *Proc. of the Doctoral Consortium of the third IEEE International Symposium on Requirements Engineering (RE' 97), Anapolis, MD, USA*, January 1997.

[HHKS94] P. Hartel, T. Hartmann, J. Kusch, and G. Saake. Specifying Information System Dynamics in TROLL. In E. Dubois, P. Hartel, and G. Saake, editors, *Proc. Workshop Formal Methods for Information System Dynamics, Utrecht (NL)*, pages 53–64. Univ. of Twente, Technical Report, 1994.

[Him97] V. Himmler. Object-Oriented Integration of User Interfaces in CATC. Diploma thesis, Technical University of Braunschweig, 1997.

[HJ94] P. Hartel and R. Jungclaus. Specifying Business Processes over Objects. In P. Loucopoulos, editor, *Proc. 13th Int. Conf. on the Entity-Relationship Approach (ER'94)*, pages 10–27. Springer, LNCS 881, Berlin, 1994.

[HJ95] P. Hartel and R. Jungclaus. Modeling Business Processes over Objects. *Int. Journal of Cooperative Information Systems*, 4(2):165–188, 1995.

[HJS92] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a Behavior Oriented Object Model. In O. Lehrmann Madsen, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP'92)*, pages 57–77. Springer, LNCS 615, Berlin, 1992.

[HJS93a] T. Hartmann, R. Jungclaus, and G. Saake. Animation Support for a Conceptual Modelling Language. In V. Mařík, J. Lažanský, and R.R. Wagner, editors, *Proc. 4th Int. Conf. on Database and Expert Systems Applications (DEXA), Prague*, pages 56–67. LNCS 720, Springer, Berlin, 1993.

[HJS93b] T. Hartmann, R. Jungclaus, and G. Saake. Spezifikation von Informationssystemen als Objektsysteme. *EMISA Forum, Mitteilungen der GI-Fachgruppe 2.5.2*, 1:2–18, 1993.

[HJSE92] T. Hartmann, R. Jungclaus, G. Saake, and H.-D. Ehrich. Spezifikation von Objektsystemen. In R. Bayer, T. Härder, and P.C. Lockemann, editors, *Objektbanken für Experten*, pages 220–242. Springer, Berlin, Reihe Informatik aktuell, 1992.

[HKSH94] T. Hartmann, J. Kusch, G. Saake, and P. Hartel. Revised Version of the Conceptual Modeling and Design Language TROLL. In R. Wieringa and R. Feenstra, editors, *Working papers of the International Workshop on Information Systems - Correctness and Reusability*, pages 89–103. Vrije Universiteit Amsterdam, 1994.

[HO71] W. Wettisch H. Olenik, H. Rentzsch. *Handbuch für den Explosionsschutz*. W.Girardet, Zürich, 1971.

[Hoh96] T. Hohnsbein. Objektorientierte Realisierung eines Meßdatenerfassungssystems für druckfeste Kapselung. Diploma thesis, Technical University of Braunschweig, 1996.

[HS93] T. Hartmann and G. Saake. Abstract Specification of Object Interaction. Informatik-Bericht 93–08, Technische Universität Braunschweig, 1993.

[HS94a] T. Hartmann and G. Saake. Prototypische Ausführung von Objektinteraktionen für die Spezifikationssprache TROLL. In F. Simon, editor, *Workshop Deklarative Programmierung und Spezifikation, Bad Honnef*, pages 56–59. Bericht Nr. 9412, Univ. Kiel, 1994.

[HS94b] T. Hohnsbein and H. Shafiee. Reengineering des Programms DRUCKMESS in der PTB. Project work, Technical University of Braunschweig, 1994.

[HSJ+94] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994.

[HW98] P. Harman and M. Watson. *Understanding UML*
The Developer's Guide . Morgan Kaufmann, USA, 1 edition, 1998.

[Jaa98] A. Jaaksi. A Method for your First Object-Oriented Project. *JOOP*, 10(8):17–25, 1998.

[JB87] H. Rechenberg J. Bortfeld, W. Hanser. *100 Jahre Physikalisch-Technische Reichsanstalt/Bundesanstalt 1887-1987*. VCH Verlagsgesllschaft, München, 1987.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999.

[JHS93] R. Jungclaus, T. Hartmann, and G. Saake. Relationships between Dynamic Objects. In H. Kangassalo, H. Jaakkola, K. Hori, and T. Kitahashi, editors, *Information Modelling and Knowledge Bases IV: Concepts, Methods and Systems (Proc. 2nd European-Japanese Seminar, Hotel Ellivuori (SF))*, pages 425–438. IOS Press, Amsterdam, 1993.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[JSHS96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.

[Jür96] G. Jürgen. Objektorientierte modellierung und prototypische implementierung des informationssystems expert an der ptb– benutzeroberfläche. Diploma thesis, Technical University of Braunschweig, 1996.

[JWH+94] R. Jungclaus, R.J. Wieringa, P. Hartel, G. Saake, and T. Hartmann. Combining TROLL with the Object Modeling Technique. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen. GI-Fachgespräch FG 1: Integration von semi-formalen und formalen Methoden für die Spezifikation von Software*, pages 35–42. Springer, Informatik aktuell, 1994.

[Kan99] M. Kananian. *Objektorientierte Erweiterung eines Informationssystemes zur Konstruktionsprüfung explosionsgeschützer elektrische Betriebsmittel.* Diploma Thesis, Technische Universität Braunschweig, May 1999.

[Kel97] W. Keller. Mapping Objects to Tables: A Pattern Language. In *Proc. of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany.* Siemens Technical Report 120/SW1/FB, 1997.

[KG98] M. Kowsari and A. Grau. An Evaluation of an Object Oriented Formal Method for Specifying Information Systems. In K. Siau, editor, *Proceedings of the Third CAiSE/IFIP 8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'98). Pisa, Italy. June 8-9, 1998*, pages M1–M12. University of Nebraska-Lincoln, USA, 1998.

[KKH+96] M. Krone, M. Kowsari, P. Hartel, G. Denker, and H.-D. Ehrich. Developing an Information System Using TROLL: an Application Field Study. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Proc. 8th Int. Conf. on Advanced Information Systems Engineering (CAiSE'96)*, pages 136–159. Springer, Berlin, LNCS 1080, 1996.

[Kow94] M. Kowsari. Entwurf eines Verwaltungssystems für Prüfvorgänge in der Physikalisch-Technischen Bundesanstalt Braunschweig. Diploma thesis, Technical University of Braunschweig, 1994.

[Kow96] M. Kowsari. Formal Object Oriented Specification Language TROLL in Information System Design. In H.-M. Haav and B. Thalheim, editors, *Doctoral Consortium of 2nd Int. Baltic Workshop on Databases and Information Systems, Tallinn, June 12-14, 1996*, 1996.

[KPLP95] B. Kitchenham, L. Pickard, and Shari. Lawrence Pfleeger. Case Studies for Method and Tool Evaluation. *The IEEE Journal*, pages 52–62, 1995.

[Küs00a] J. Küster Filipe. *Foundations of a Module Concept for Distributed Object Systems.* PhD Thesis, Technical University Braunschweig, September 2000.

[Küs00b] J. Küster Filipe. Fundamentals of a Module Logic for Distributed Object Systems. *Journal of Functional and Logic Programming*, 2000(3), March 2000.

[Lan95] K. Lano. *Formal Object-Oriented Development.* Springer, London, 1995.

[Let99] P. O. Letelier Torres. *Animación Automática de Especificaciones OASIS utilizando Programación Lógica Concurrente.* PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 1999.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics.* Prentice-Hall, Inc., New Jersy, 1994.

[LSR97] P. Letelier, P. Sánchez, and I. Ramos. Animation of System Specifications using Concurrent Logic Programming. In *Symposium of Logical Approaches to Agent Modeling and Design (ESSLLI'97), Aix-in-Provence (France)*, 1997.

[LSR99] P. Letelier, P. Sánchez, and I. Ramos. Animation of Conceptual Models using Two Concurrent Environments: An Overview. In *Proc. of the 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communication and Computers, Greece*, 1999.

[MC90] S.L. Meira and A.L.C. Cavalcanti. Modular Object-Oriented Z Specifications. In *Z User workshop, Oxford*. Springer-Verlag, 1990.

[Moh94] K. Mohammady. Entwurf eines informationssystems zur konstruktionsprüfung explosionsgeschützter betriebsmittel. Diploma thesis, Technical University of Braunschweig, 1994.

[MS92] JD. McGregor and DA. Sykes. *Object-Oriented Software Development: Engineering Software for Reuse*. International Thomson computer Press, Boston, MA, 1992.

[ORW83] H. Olenik, H. Rentzsch, and W. Wettstein. *Explosion Protection Manual*. W. Girardet, Essen, 2nd revised edition, 1983.

[PCR99] O. Pastor, J. H. Canós, and I. Ramos. From CASE to CARE (Computer-Aided Requirements Engineering). In J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, editors, *Proc. of the 18th Int. Conference on Conceptual Modeling (ER'99), Paris (France)* , pages 278–292, November 1999.

[PE01] R. Pinger and H.-D. Ehrich. Compositional Checking of Communication among Observers. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE), Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2001), Genova*, volume 2029 of *Lecture Notes in Computer Science*, pages 32–44, 2001.

[PIP+97] O. Pastor, E. Insfran, V. Pelechano, J. Romero, and J. Merseguer. OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods. In A. Olive and J. A. Pastor, editors, *Proc. of the 9th International Conference on Advanced Information Systems Engineering (CAiSE'97), Barcelona*, pages 145–158, 1997.

[PPIG98] O. Pastor, V. Pelechano, E. Insfrán, and J. Gómez. From Object-Oriented Conceptual Modeling to Automated Programming in Java. In T.W. Ling, S. Ram, and M.L. Lee, editors, *Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore* , pages 183–196. Springer, LNCS 1507, November 1998.

[RB97] J. Robie and D. Bartels. A Comparison between Relational and Object-Oriented Databases for Object Oriented Application Development. White Paper, POET Software Corp., 1997. Available at http://www.poet.com/.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, Reading, MA, 1999.

[Rum94] J. Rumbaugh. Waht is a Method. *Journal of Object-Oriented Programming*, 23(1):81–85, Mai. 1994.

[Saa96] A. Saad. Objektorientierte Realisierung der Benutzerschnittstellen eines informationsystems zur verwaltung der prüfvorgängein der ptb. Diploma thesis, Technical University of Braunschweig, 1996.

[Sch96a] M. Schönhoff. Objektorientierte Realisierung eines Steuerungs- und Überwachungssystems für Explosionsprüfstände. Diploma thesis, Technical University of Braunschweig, 1996. Available on http://www.ifi.unizh.ch/~mschoen.

[Sch96b] Schwarz. Objektorientierte spezifikation eines informationssystems für kunststoffe in explosionsgeschützten elektrischen betriebsmitteln. Diploma thesis, Technical University of Braunschweig, 1996.

[Sch00] C. Schmidt. *Remodellierung, Neuimplementierung und Erweiterung eines Meßdatenbearbeitungssystems für druckfeste Kapselung.* Diploma Thesis, Technische Universität Braunschweig, February 2000.

[SFSE88] A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. Abstract Object Types: A Temporal Perspective. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Colloq. on Temporal Logic in Specification*, pages 324–350. LNCS 398, Springer, Berlin, 1988.

[SH94] G. Saake and T. Hartmann. Modelling Information Systems as Object Societies. In K. von Luck and H. Marburger, editors, *Management and Processing of Complex Data Structures, Proc. 3rd Workshop on Information Systems and Artificial Intelligence, Hamburg*, pages 157–180. Springer, Berlin, LNCS 777, 1994.

[Sha96] H. Shafiee. Objektorientierte Realisierung der Benutzerschnittstellen eines Meßdatenbearbeitungssystems für druckfeste Kapselung. Diploma thesis, Technical University of Braunschweig, 1996.

[SHJ+94] G. Saake, P. Hartel, R. Jungclaus, R. Wieringa, and R. Feenstra. Inheritance Conditions for Object Life Cycle Diagrams. In U. Lipeck and G. Vossen, editors, *Workshop Formale Grundlagen für den Entwurf von Informationssystemen, Tutzing*, pages 79–89. Technical Report Univ. Hannover, No. 03/94, 1994.

[SHJE94] G. Saake, T. Hartmann, R. Jungclaus, and H.-D. Ehrich. Object-Oriented Design of Information Systems: `TROLL` Language Features. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*, pages 219–245. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 1994.

[SJ92] G. Saake and R. Jungclaus. Views and Formal Implementation in a Three-Level Schema Architecture for Dynamic Objects. In P.M.D. Gray and R.J. Lucas, editors, *Advanced Database Systems : Proc. 10th British National Conference on Databases (BNCOD 10), July 6-8, 1992, Aberdeen (Scotland)*, pages 78–95. Springer, LNCS 618, Berlin, 1992.

[SJE92] G. Saake, R. Jungclaus, and H.-D. Ehrich. Object-Oriented Specification and Stepwise Refinement. In J. de Meer, V. Heymer, and R. Roth, editors, *Proc. Open Distributed Processing, Berlin (D), 8.-11. Okt. 1991 (IFIP Transactions C: Communication Systems, Vol. 1)*, pages 99–121. North-Holland, 1992.

[SJH93a] G. Saake, R. Jungclaus, and T. Hartmann. Application Modelling in Heterogeneous Environments using an Object Specification Language. In M. Huhns, M.P. Papazoglou, and G. Schlageter, editors, *Int. Conf. on Intelligent & Cooperative Information Systems (ICICIS'93)*, pages 309–318. IEEE Computer Society Press, 1993.

[SJH93b] G. Saake, R. Jungclaus, and T. Hartmann. Application Modelling in Heterogeneous Environments using an Object Specification Language. *Int. Journal of Intelligent and Cooperative Information Systems*, 2(4):425–449, 1993.

[SK97] M. Schönhoff and M. Kowsari. Specifiying the Remote Controlling of Valves in an Explosion Test Environment. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe, FME'97, 4th Intern. Symposium, Technical University Graz, Austria, 15-19 September, 1997*, pages 201–220. Springer, Berlin, LNCS 1313, 1997.

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.

[Ste94] W. Stein. *Objektorientierte Analysemethoden Vergleich, Bewertung, Auswahl.* BI-Wiss.-Verl., 1nd edition, 1994.

[TD96] R.H. Thayer and M. Dorfman. *Software Requirements Engineering.* IEEE Computer Society, Los Alamitos, 1996.

[Tha94] G.E. Thaller. *Software-Metriken - einsetzen, bewerten, messen.* Heinz Heise, Hannover, 1994.

[Wes02] J. Christopher Westland. The cost of errors in software development: evidence from industry. *Journal of Systems and Software*, 62(2):85–96, 2002.

[Win00] C. Winter. *Remodellierung, Neuimplementierung und Erweiterung eines Meßdatenerfassungssystems für druckfeste Kapselung.* Diploma Thesis, Technische Universität Braunschweig, February 2000.

[WJH+93] R. Wieringa, R. Jungclaus, P. Hartel, T. Hartmann, and G. Saake. OMTROLL – Object Modeling in TROLL. In U.W. Lipeck and G. Koschorreck, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93*, pages 267–283, 1993.

[Wol98] E. Wolter. Entwurf und implementierung eines parsers für vereinfachte sql-anfragen einer meßsdatenban, 1998.

[Woz98] E. Wozniak. Objektorientierte softwaremetriken für TROLL- spezifikationen und deren c++ -implementierungen. Diploma thesis, Technical University of Braunschweig, 1998.

[XJG98] Y. Xiaodong, C. Jiajun, and Z. Guoliang. Two-dimensional Software Development Model Combining Object-Oriented Method with Formal Method. *Software Engineering*, 23(1):81–85, January. 1998.

[ZH94] J.-M. Zeippen and P. Hartel. Specification of a Control System by Domain Specialists with OBLOG – Experience Report –. In E. Dubois, P. Hartel, and G. Saake, editors, *Proc. Workshop Formal Methods for Information System Dynamics, Utrecht (NL)*, pages 137–146. Univ. of Twente, Technical Report, 1994.

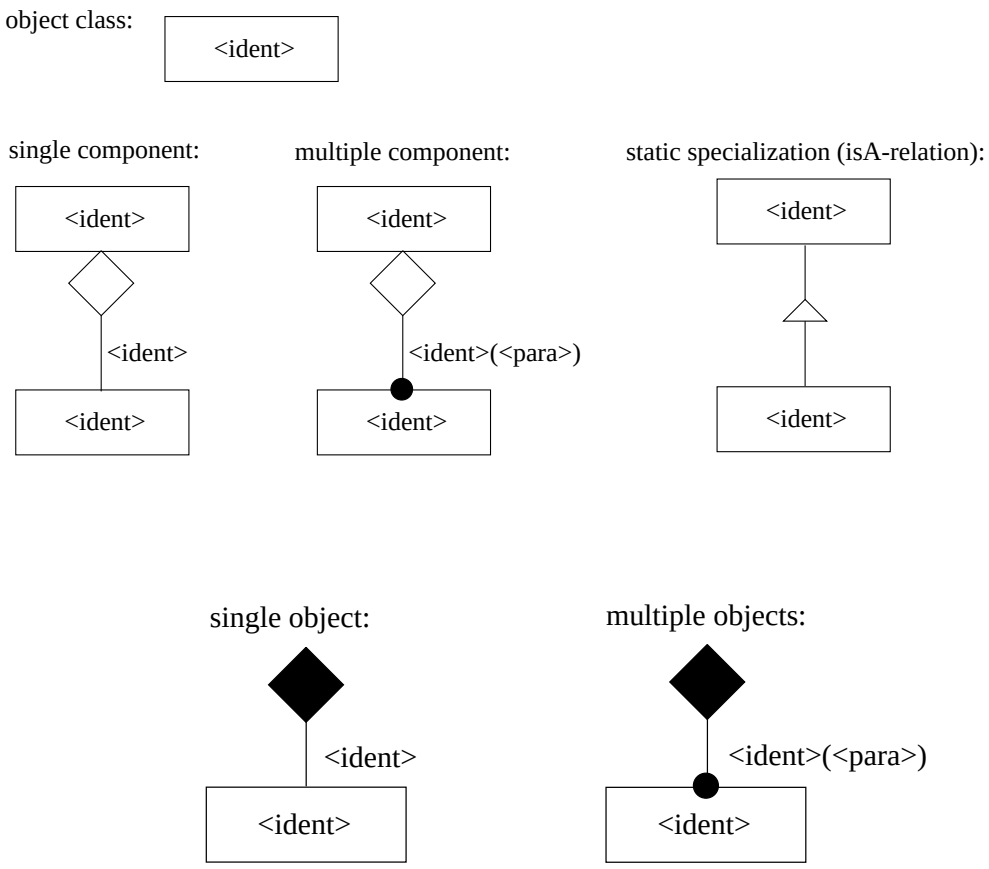
Appendix A

Syntax

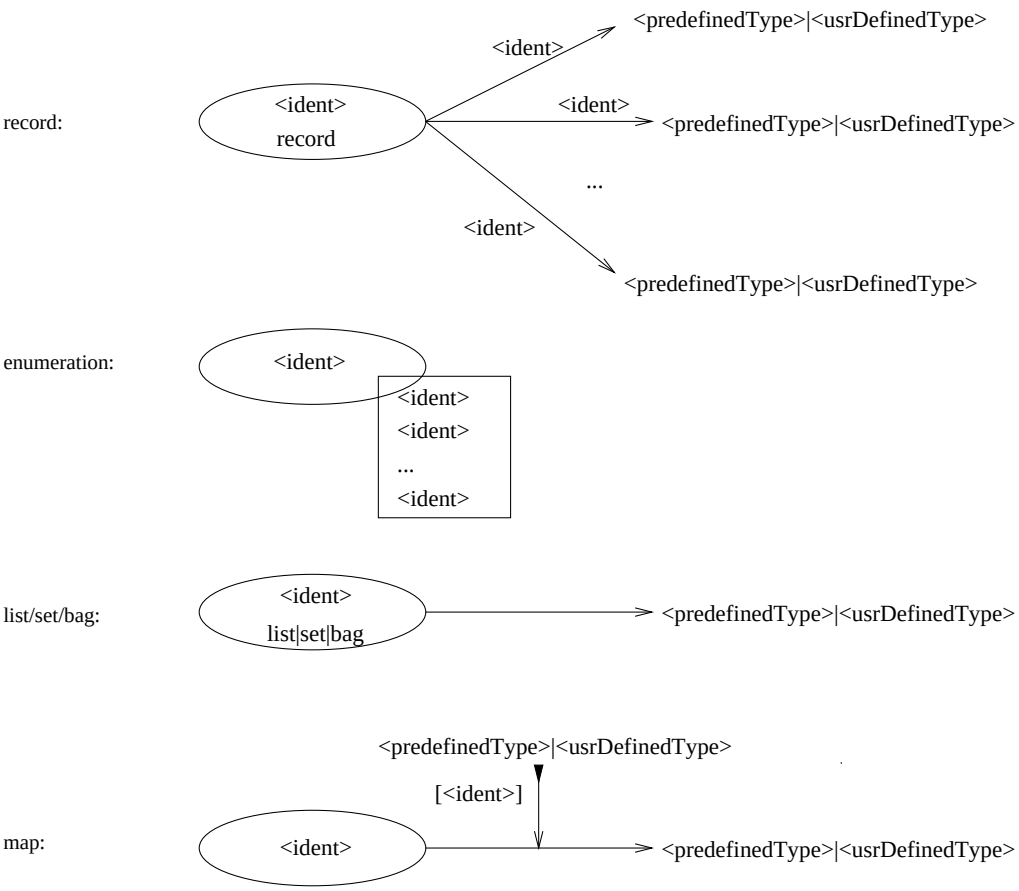
This appendix contains all OMTroll diagrams and the entire Troll syntax.

A.1 OMTroll

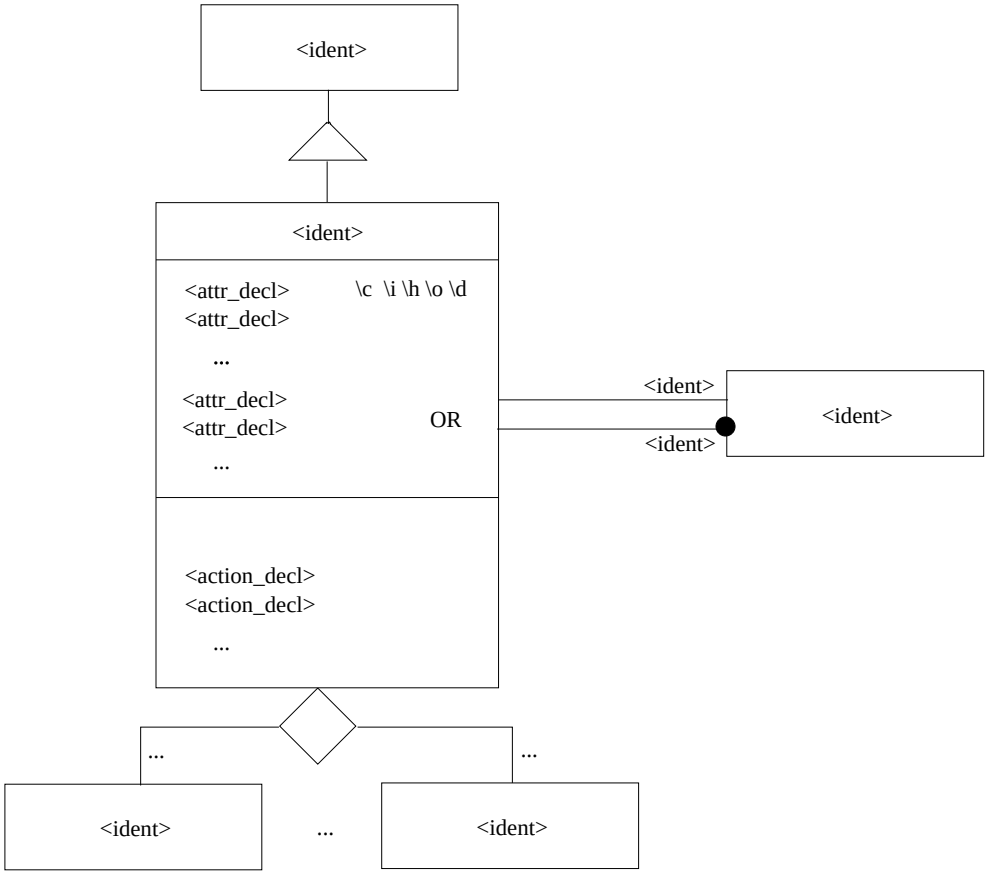
community diagram



data type diagram



object class declaration diagram



object behavior diagram

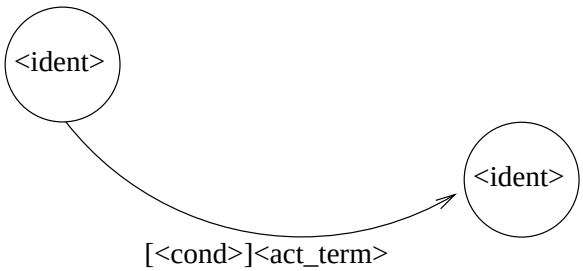
initial state:

final state:

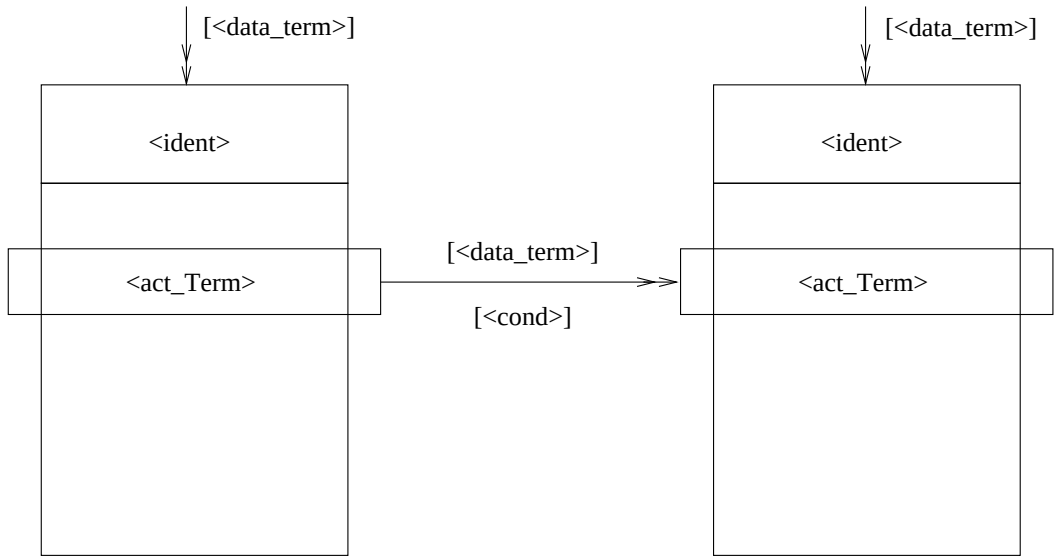
state:

<ident>

transition: <variables>



communication diagram



A.2 TROLL

reserved words

data type	*var*	*object class*	*end*	*attributes*	*actions*
components	*behavior*	*aspect of*	*false*		*hidden*
derived	*constant*	*optional*	*initialized*	*once*	*onlyIf*
do	*od*	*if*	*then*	*else*	*fi*
forEach	*constraints*	*initially*	*objects*	*object system*	

predefined names

nat	*int*	*real*	*string*	*bool*	*char*	*money*	*date*	*list*
set	*bag*	*record*	*map*	*in*	*or*	*and*	*xor*	*implies*
not	*div*	*mod*	*toSet*	*toList*	*toBag*	*cnt*	*tail*	*head*
dom	*range*	*def*	*select*	*from*	*where*	*all*	*any*	*enum*

meta rules

<*x_list*> ::= <*x*> | <*x*> , <*x_list*>

<*x_seq*> ::= <*x*> | <*x*> ; <*x_seq*>

identifier

<*ident*> ::= <*character*> | <*character*><*ident*>

<*character*> ::= A | ... | Z | a | ... | z | 0 | 1 | ... | 9 | _

data types

<*type*> ::= <*ident*> | |<*ident*> | | enum(<*ident_list*>) |
set(<*type*>) | list(<*type*>) | bag(<*type*>) |
record(<*field_list*>) | map(<*field*> , <*type*>) |
nat | int | real | bool | string | date | money | char

<*dataTypeSpec*> ::= data type <*ident*> = <*type*>

<*field*> ::= <*variable*> | <*type*>

variables

<*variableDecl*> ::= var <*variable_list*>

<*variable*> ::= <*ident*> : <*type*>

terms

data terms

<*constTerm*> ::= <*ident*> | <*natConst*> | <*intConst*> | <*realConst*> |
<*stringConst*> | <*charConst*> | <*boolConst*> |
<*dateConst*> | <*moneyConst*> | empty

<*mapLet*> ::= (<*dataTerm*> ,<*dataTerm*>)

<*constructor*> ::= mk-<*type*> ([<*dataTerm_list*>]) | mk-<*type*> ([<*mapLet_list*>])

<*relation*> ::= < | > | <= | >= | = | # | in

<*boolOp*> ::= or | and | implies | xor

<*infixOp*> ::= + | - | * | / | div | mod | .@ | .. | isA | <*boolOp*> |
<*relation*>

<*prefixOp*> ::= - | head | tail | cnt | toSet | rng | dom | def | num

<*condTerm*> ::= <*pFormula*> ? <*dataTerm*> : <*dataTerm*>

<*selectTerm*> ::= select <*dataTerm*> from <*rangeDecl*> [where <*pFormula*>]

<*dataTerm*> ::= <*ident*> | <*qualidentTerm*> | <*constTerm*> |
(<*dataTerm*>) | <*prefixOp*> (<*dataTerm_list*>) |
<*constructor*> | <*dataTerm*> <*infixOp*> <*dataTerm*>|
<*condTerm*> | <*selectTerm*> | <*dataTerm*> (<*dataTerm*>) |
<*dataTerm*> .<*dataTerm*>

qualified identifier terms

<*qualidentTerm*> ::= <*ident*> | <*componentTerm*> .<*ident*>

component terms

<*componentTerm*> ::= <*ident*>[(<*dataTerm*>)] |
<*ident*>[(<*dataTerm*>)].<*componentTerm*>

action terms

<*actionTerm*> ::= <*ident*>[(<*dataTerm_list*>)] |
<*componentTerm*> .<*ident*>[(<*dataTerm_list*>)]

range declaration

<*rangeDecl*> ::= <*ident*> in <*dataTerm*> |
<*ident*> in <*dataTerm*>, <*rangeDecl*>

propositions

<*pFormula*> ::= not <*pFormula*> | <*pFormula*> <*boolOp*> <*pFormula*> |
<*dataTerm*> | all<*rangeDecl*> (<*pFormula*>) |
(<*pFormula*>) | any<*rangeDecl*> (<*pFormula*>)

object classes

<*objectClassSpec*> ::= object class <*ident*>
[<*specialization*>]
[components<*componentDecl_seq*> ;]
[<*signatureDecl*>]
[behavior <*behaviorDef*>]
end

<*specialization*> ::= aspect of <*ident*> if <*specCondition_list*>

<*specCondition*> ::= <*actionTerm*> [and <*pFormula*>]

signature declaration

<*signatureDecl*> ::= [attributes<*attributeDecl_seq*> ;]
[actions<*actionDecl_seq*> ;]

<*attributeDecl*> ::= <*variable*> [<*attributeDesc_list*>]

<*attributeDesc*> ::= hidden | constant | optional |
derived <*dataTerm*> | initialized <*constTerm*>

<componentDecl> ::= <ident>[(<field>)] :<ident> [once] [hidden]

<actionDecl> ::= [*| +] <actionSignature> [hidden]

<actionSignature> ::= <ident> [(<parameter_list>)]

<parameter> ::= [!]<field>

behavior definition

<behaviorDef> ::= [<operationDef_seq> ;]
[<constraintDef> ;]

<operationDef> ::= <actionTerm>
[onlyIf<pFormula>]
[<variableDecl>]
[do <actionRule> od]

<actionRule> ::= <actionRule_list> | <basicRule> |
<repetitiveRule> | <conditionalRule>

<conditionalRule> ::= if <pFormula> then <actionRule> [else <actionRule>] fi

<repetitiveRule> ::= forEach<rangeDecl> do <actionRule> od

<basicRule> ::= <valuation> | <actionCall>

<valuation> ::= <ident> := <dataTerm>

<actionCall> ::= <actionTerm>

<constraintDef> ::= constraints <constraintRule_list>

<constraintRule> ::= <pFormula> | initially <pFormula>

system specification

<systemSpec> ::= object system <ident>
<specItem_seq>
end.

<specItem> ::= <dataTypeSpec> | <objectClassSpec> |
<instanceDecl> | <behaviorSpec>

<instanceDecl> ::= objects <ident>[(<field>)] :<ident> [once]

<behaviorSpec> ::= behavior <operationDef_seq> end

Appendix B

TROLL Example

This appendix contains the TROLL specification of the CATC example introduced in Chapter 4.

```
/* Data type definitions */
data type labours = enum(l3_41, l3_42, l3_43);
data type msset = record(press:real, time:real);
data type msresults = list(real);
data type address_type = record(street:string,nr:nat,city:string);
data type users_type = enum(admin, staff, operator);

/* Object class Group */
object class Group
  components
    Applications(appNr:nat) : Application;
    Companies(compId:nat) : Company;
  actions
    *create;
    +delete;
end;

/* Object class Application */
object class Application
  components
    Experiments(expNr:nat) : Experiment;
  attributes
    company : |Company| constant;
    labour : labours;
    max_pressure : real constant;
    nextExpNr : nat initialized 1, hidden;
  actions
    *createAppl(comp:|Company|,max_press:real,lab:labours);
    newExperiment(nam:string,st:msset,!expNr:nat);
    +deleteAppl;
  behavior
    createAppl(comp,max_press,lab)
    do
```

```
      company := comp,
      max_pressure := max_press,
      labour := lab
    od;
    newExperiment(nam,st,expNr)
    onlyIf(st.press <= max_pressure)
    do
      Experiments(nextExpNr).createExp(nam,st),
      expNr := nextExpNr,
      nextExpNr := nextExpNr+1
    od;
  constraints
    nextExpNr <= 11;
end;

/* Object class Experiment */
object class Experiment
  attributes
    name : string constant;
    setup : msset constant;
    results : msresults initialized empty;
    assessments : list(string) initialized empty;
  actions
    *createExp(nam:string,st:msset);
    giveSetup(!st:msset);
    storeResults(res:msresults);
    giveResults(!res:msresults);
    storeAssessments(assmt:list(string));
    deleteExp;
  behavior
    createExp(nam,st)
    do
      name := nam,
      setup := st
    od;
    giveSetup(st)
    do
      st := setup
    od;
    storeResults(res)
    do
      results := res
    od;
    giveResults(res)
    do
      res := results
    od;
    storeAssessments(assmt)
    do
      assessments := assmt
```

```
    od;
end;

/* Object class Company */
object class Company
  attributes
    name : string;
    address : address_type;
    phone : string;
  actions
    *create(nam:string,addr:address_type,phon:string);
    +delete;
  behavior
    create(nam,addr,phon)
    do
      name := nam,
      address := addr,
      phone := phon
    od;
end;

/* Object class User */
object class User
  attributes
    shortName : string constant;
    login_date : date constant;
  actions
    *login(n:string,d:date,user_t:users_type);
    +logout;
  behavior
    login(n,d,user_t)
    do
      shortName := n,
      login_date := d
    od;
end;

/* Object class Staff */
object class Staff
  aspect of User if login(n,d,t) and  t = staff
  actions
    createExperiment(appNr:nat,nam:string,st:msset,!expNr:nat);
    askResults(appNr:nat,expNr:nat,!res:msresults);
    checkResults(appNr:nat,expNr:nat,res:msresults);
    giveAssessment(appNr:nat,expNr:int,assmt:list(string));
end;

/* Object class Operator */
object class Operator
  aspect of User if login(n,d,t) and  t = operator
```

```
  actions
    askSetup(appNr:nat,expNr:nat,!st:msset);
    startExperiment(appNr:nat,expNr:nat,st:msset);
    giveResults(appNr:nat,expNr:int,res:msresults);
end;

/* Object declarations */
objects IG34:Group;
objects Users(userId:nat):User;

/* Global behavior specification */
behavior
  Staff(Users(userId)).createExperiment(appNr,nam,st,expNr)
  do
    IG34.Applications(appNr).newExperiment(nam,st,expNr)
  od;
  Staff(Users(userId)).askResults(appNr,expNr,res)
  do
    IG34.Applications(appNr).Experiments(expNr).giveResults(res)
  od;
  Staff(Users(userId)).giveAssessment(appNr,expNr,assmt)
  do
    IG34.Applications(appNr).Experiments(expNr) \
    .storeAssessments(assmt)
  od;
  Operator(Users(userId)).askSetup(appNr,expNr,st)
  do
    IG34.Applications(appNr).Experiments(expNr).giveSetup(st)
  od;
  Operator(Users(userId)).giveResults(appNr,expNr,res)
  do
    IG34.Applications(appNr).Experiments(expNr).storeResults(res)
  od;
end;
```

Figure C.1: OWL-Classes

Appendix C

Object Windows Library

This appendix give an overview about Object Windows Library.

TWindow Classes

Application Class: TApplication

The class TApplication builds on the basic class of Windows programming and represents a complete Windows application. However, this class is never directly used; a derived class is defined which is supplied by the required functionalities.

Window Class: TWindow

Among the class Tapplication, the class TWindow builds on the second central class for Windows programming. This class is compatible with the different switches and dialogue boxes used in Windows and also builds the basic class for all Window types.

Dialog Class: TDialog

The class TDialog derived from TWindow enables the creation, the realisation and the removal of dialogue. Such dialogue windows are mainly used for making dialogue input by the user possible.

Control Element Classes

The quantity of window control elements is comprised mainly of input fields and control boxes. This quantity comes from the basic class TControl. Various different classes are derived from this class which defines different control elements. The following list gives an insight into all used control elements of the implementation of the information system "Operation"

Tbutton

This class represents a control box which resolves an action of the corresponding dialogue.

TcheckBox

This class was defined for the use of marking fields checkboxes.

TRadioButton

This class makes the administration of RadioButton possible which enable a "l or n" choice of different data.

TGroupBox

By framing groups, several control elements may be combined to one or several groups within the dialogues.

TStatic

This class is used for output fields for labelling various control elements.

TEdit

This class is used for the use of input fields.

TListBox

This class makes the administration of list of elements possible.

TComboBox

This class is for presenting combination fields which are an extension of listing fields. They combine a static textual element or an edit field with a list box.

Class Communication

The user interfaces developed in this thesis is a compilation of several classes which derive from the OWL class.

The self-defined classes must communicate with each other to make an interactive operation of this programme possible. Communication is carried out by exchanging messages and information between the different classes.

This exchange is realised by the so-called response tables of ObjectWindows. Response tables build detailed messages to corresponding response functions. The declaration of response tables occurs in the relevant class declaration by means of the following macro:

DECLARE$_{R}ESPONSE_{T}ABLE$

The definition of the response table is initiated by the following macro:

DEFINE$_{R}ESPONSE_{T}ABLE$

The value of "X" depends on the quantity of the direct basic classes of the relating class of the response table. The definition of the response table is ended by the macro:

END$_{R}ESPONSE_{T}ABLE$

Appendix D

C++ Classes of Remote of Valve

This appendix give results of the C++ Implementaion of *Valve*.

Class	Attribute		NOM	LOC	Datendefinition	
	obj.valued	simple			Constr.	simple.
CvtAktion	0	1	5	14	0	0
CvtAnschluss	3	3	17	57	0	6
CvtAnschlussPosition	0	1	5	5	0	0
CvtGasflussschema	1	1	9	21	0	2
CvtKnoten	9	4	51	207	12	43
CvtActivKnots	2	2	16	84	6	11
CvtGMG	1	3	10	120	5	15
CvtHandValve	0	1	9	39	0	3
CvtMESG	1	2	7	37	1	1
CvtPumpe	0	3	13	91	3	6
CvtDryer	0	2	12	60	0	6
CvtValve	0	1	12	102	0	9
CvtPassivKnots	0	0	11	9	0	0
CvtBerstsingcontrol	0	0	3	44	0	8
CvtPresssensor	0	1	4	33	0	2
CvtCoverOutside	0	0	4	72	4	11
CvtBoilerExEva	0	0	4	57	4	9
CvtBoilerHalle	0	1	6	70	6	9
CvtPipeEnd	1	0	4	68	0	3
CvtPipeConnection	0	0	3	38	0	3
CvtOxigenanalyse	0	0	3	2	0	1
CvtStorage	0	0	3	42	3	8
CvtKnotsReferenz	0	2	8	12	0	0
CvtShortform	0	1	5	24	0	0
CvtDirektion	0	1	3	3	0	0
CvtPfeil	2	0	4	23	3	4
Cvtduty	3	3	11	30	0	1
CvttimeDuty	1	0	5	8	0	0
CvtDirektion	0	1	4	6	0	0
CvtKeyWord	1	0	10	104	0	2
Cvtfile	2	0	4	15	1	2
Cvttime	0	1	4	4	0	0
CXvtExeption	0	1	3	15	0	0
CXvtNoACtion	1	0	2	3	0	0
CXvtNoConnectionPosition	0	1	2	3	0	0
CXvtNoDireCtion	0	1	2	3	0	0
CXvtGasNotcomplete	0	0	2	2	0	0
CXvtFileNotfind	1	0	2	3	0	0
CXvtToomuchText	1	0	2	3	0	0
CXvtShortformMuliple	1	0	2	3	0	0
CXvtKnotsNotCalculate	2	0	2	4	0	0
CXvtConnectionCovered	4	0	2	6	0	0
CXvtConnectionNotDefined	2	0	2	4	0	0
CXvtKnotsNotExist	1	0	2	3	0	0
CXvtPipeTwoKnots	0	0	2	2	0	0
CXvtPipeWithoutGasflow	0	0	2	2	0	0

Class	Attribute		NOM	LOC	Data definition	
	obj.valued	simple			Constr.	simple
CXvtPipeSlop	0	0	2	2	0	0
CXvtPipeNotCalculated	1	0	2	3	0	0
CXvtPipeFromToNotCalculated	5	0	2	7	0	0
CXvtTextNotCalculated	1	0	2	3	0	0
CXvtPfeilNotCalculated	1	0	2	3	0	0
CXvtDelayTooLang	0	2	2	4	0	0
CXvtNoRestor	1	0	2	3	0	0
CXvtDontfillduty	1	0	2	3	0	0
CXvtdutyNotCalculated	1	0	2	3	0	0
CXvtdutyForNotCalculated	2	0	2	4	0	0
CXvtNoWindow	0	0	2	2	0	0
CXvtConnectionMultiple	1	0	2	2	0	0
CXvtNotallowedAPos	2	0	2	4	0	0
CXvtNoACtivShortform	1	0	2	3	0	0
CXvtNoPassiveShortform	1	0	2	3	0	0
CXvtNoStartbar	1	0	2	3	0	0
CXvtHereNoDirection	1	0	2	2	0	0
CXvtNoOffOnClose	1	0	2	3	0	0
CXvtNoOnOffPressure	1	0	2	3	0	0
CXvtPipeverbDirektion	0	0	2	2	0	0
CXvtGMGwithoutAir	0	0	2	2	0	0
CXvtNo Shortform	1	0	3	8	0	0
CXvtShortformAnfangAktiv	0	1	2	3	0	0
CXvtShortformMitS	0	2	2	5	0	0
CXvtNoPfeil	0	0	2	2	0	0
CXvtNoPosition	0	0	2	2	0	0
CXvtNoKeyword	1	0	2	2	0	0
CXvtNoTimepoint	1	0	2	3	0	0
SvtGasInfo[1]	0	3	3	9	0	0
SvtGMGInfo[1]	0	4	1	5	0	0
TApplication[2]						
TvtVentilApp	0	2	2	14	0	1
TArrayAsVector[2]						
CvtConnectionListe	0	0	1	1	0	0
TArrayAsVectorIterator[2]						
CvtConnectionListeIter	0	0	1	1	0	0
TBinarySearchTreeImp[2]						
CvtKnotsListe	0	0	1	1	0	0
TBinarySearchTreeIteratorImp[2]						
CvtKnotsListeIter	0	0	1	1	0	0
TDecoratedFrame[2]						
TvtVentilFrame	4	8	33	253	4	17
TDialog[2]						
TvtDialog	0	0	2	17	0	0
TvtDlgGMG	0	14	14	120	4	18
TvtDlgMESG	0	2	5	21	0	4
TIListImp[2]						
CvtThreadListe	0	0	5	19	2	2
CvtGfThreadListe	0	0	1	1	0	0

Class	Attribute		NOM	LOC	Data definition	
	obj.valued	simple			Constr.	simple
TIListIteratorImp[2]						
CvtThreadListeIter	0	0	1	1	0	0
CvtGfThreadListeIter	0	0	1	1	0	0
TISetAsVector[2]						
CvtKnotsAmounts	0	0	7	36	8	0
TISetAsVectorIterator[2]						
CvtKnotsAmountsIter	0	0	1	1	0	0
TPoint[2]						
TvtPosition	0	0	4	4	0	0
TPXPictureValidator[2]						
TvtSpaltValidator	0	0	2	2	0	0
TRadioButton[2]						
TvtRadioButton	0	0	2	11	0	1
TSet[2]						
CvtKnotsAmountsAmounts	0	0	4	18	4	0
TSetAsVector[2]						
CvtPfeilAmounts	0	0	1	1	0	0
Cvtduty Amounts	0	0	3	9	2	0
CvtTextAmounts	0	0	1	1	0	0
CvtZeitduty Amounts	0	0	1	1	0	0
TSetAsVectorIterator[2]						
CvtPfeilAmountsIter	0	0	1	1	0	0
Cvtduty AmountsIter	0	0	1	1	0	0
CvtTextAmountsIter	0	0	1	1	0	0
Cvttimeduty AmountsIter	0	0	1	1	0	0
TSetIterator[2]						
CvtKnotsAmountsAmountsIter	0	0	1	1	0	0
TSlider[2]						
TvtSlider	0	0	3	13	0	0
TStatusBar[2]						
TvtStatBar	0	2	15	61	1	3
TThread[2]						
CvtGasflowThread	0	1	5	13	0	1
TWindow[2]						
TvtGraphic	5	6	34	262	15	51

[1] Structur
[2] Classes from Class Library

Following Functions belong to the Project, are defined in other classes.

Funktion	LOC	Data definition	
		Constr.	simple
istream& operator>> (istream&, CvtConnectionPosition&)	8	0	1
ostream& operator<< (ostream&, CvtConnectionPosition&)	1	0	0
istream& operator>> (istream&, CvtDirection&)	8	0	1
ostream& operator<< (ostream&, CvtDirection&)	1	0	0
ostream& operator<< (ostream&, CXvtExeption&)	2	0	0

Funktion	LOC	Data definition	
		Constr.	simple
static void Change (void *) (in *Drucksen.cpp*)	1	0	0
static void Change (void *) (in *Gmg.cpp*)	1	0	0
istream& operator>> (istream&, CvtGasflowschema::CvtShortformAmounts&)	9	0	1
ostream& operator<< (ostream&, CvtGasflowschema::CvtShortformAmounts&)	5	0	0
istream& operator>> (istream&, CvtGasflowschema&)	10	0	0
ostream& operator<< (ostream&, CvtGasflowschema&)	5	0	0
static void __syscall Execute (void *)	1	0	0
static void Change (void *) (in *Mesg.cpp*)	1	0	0
static const char* SpaltInString (MESG_SPALT)	2	1	1
static MESG_SPALT StringInSpalt (const char *)	1	0	0
istream& operator>> (istream&, SvtGasInfo&)	7	0	2
static void Change (void *) (in *Handvent.cpp*)	1	0	0
static void Change (void *) (in *Kesselhl.cpp*)	1	0	0
CvtKnotsAmounts operator+ (const CvtKnotsAmounts&, CvtKnotsAmounts&)	7	2	1
ostream& operator<< (ostream&, const CvtKnotsAmounts&)	7	0	0
CvtKnotsAmountsAmounts operator/ (const CvtKnotsAmountsAmounts&, CvtKnotsAmountsAmounts&)	8	1	0
ostream& operator<< (ostream&, const CvtKnotsAmountsAmounts&)	8	2	1
istream& operator>> (istream&, CvtShortform&)	11	0	1
ostream& operator<< (ostream&, CvtShortform&)	5	0	0
istream& operator>> (istream&, CvtPfeil&)	4	0	1
istream& operator>> (istream&, TvtPosition&)	14	0	2
ostream& operator<< (ostream&, TvtPosition&)	1	0	0
static void ChangeEnergie (void *)	1	0	0
static void ChangeVentile (void *)	1	0	0
istream& operator>> (istream&, CvtKeyWord&)	5	0	0
ostream& operator<< (ostream&, CvtKeyword&)	1	0	0
istream& operator>> (istream&, CvtTexte&)	3	0	0
ostream& operator<< (ostream&, CvtTexte&)	1	0	0
ostream& operator<< (ostream&, CvtTexteAmounts&)	8	1	0
int OwlMain (int, char* [])	16	0	0
static void Change (void *)	1	0	0
istream& operator>> (istream&, CvtTimepoint&)	8	0	1